UNIVERSITY OF OSLO
Department of Informatics

Improving Inter-subdomain Communication and Load-balancing for the Parallel Diffpack Library

Master's thesis

Martin Burheim Tingstad

June 21, 2007

Contents

Chapter 1

Introduction

1.1 Background

Matrix and vector operations are a substantial part of the scientific computing workload, and have been subject to much work and many optimisations in order to increase the performance and efficiency. The introduction of parallel computing and distributed data has complicated the work required to achieve gains in performance, and several libraries have been written in an attempt to hide much of the details and the difficulty involved with high-performance parallel programming. With parallel computing came many new concepts and challenges to computer science. For example the gain by using several processors to do the work previously done by one was defined as *speedup* (Equation 1.1), where T_p is the time it takes the parallel implementation to complete the same task (On p processors) as the serial implementation can do in time T_s.

$$S_p = \frac{T_s}{T_p} \tag{1.1}$$

Speedup equal to p (The number of processors) is called linear speedup, and implies that doubling the number of processors halves the wall-time required to complete the specific task. This is considered good speedup, but is difficult to achieve due to the added communication between the processors in addition to the computations they had to do initially. Sometimes super-linear speedup ($S_p > P$) is observed, as splitting the domain over several processors will make the sub-domains fit in a higher level of cache at the processors. One industry standard library for inter-processor communication is the *Message Passing Interface* (MPI [1]). To minimise the effects of communication it supports several communication methods, deciding which one is best suited depends on the situation.

1.1.1 Scope of Problem

We have concentrated our efforts on improving typical numerical methods for solving PDEs using domain decomposition methods[2]. Using domain decomposition methods generate boundaries with or without overlaps. Iterative solvers require that the data along the boundaries (and the overlap) must be

exchanged between the processors at every iteration, and it is crucial that this is done as efficiently as possible. An increase in communication will always have a negative impact on speedup, as there is overhead associated with starting and stopping the communication at the processors, in addition to the cost of crossing a relatively slow interconnect. Speedup is also limited by how well the amount of work is distributed across the processors. If all p processors in the system must finish with the data they have been allocated before they can continue (As is the case for iterative solvers), an uneven distribution of work will lead to more waiting and a lower speedup.

1.2 Our work

In this report we attempt to alleviate the two obstacles to speedup we have mentioned. The new functionality has been implemented in Diffpack[3], and the improvements have been measured using libraries that can trace the execution of MPI-programs. To find libraries that can do this easily, several have been tested. These libraries are presented in Chapter 2.

1.2.1 Overlapping communication and computation

MPI supports several types of communication, where overhead associated with these different types will affect the program differently depending on call pattern and frequency. Different properties here are blocking, buffering and polling etc. For an in-depth discussion see [4]. Non-blocking communication functions return immediately, even though communication might not be finished. This allows the processors to for example do useful computations between calling the non-blocking calls to start and finish the communication; Hence the term overlapping communication and computations.

As long as the time spent on communication is less than the computations required to be done at the same time, this should completely hide the time it takes the data to cross the interconnect, and reduce the performance penalties communication is usually associated with. Our initial feasibility study testing overlapping of communication and computation using explicit solvers for wave equations will be described in Chapter 3, and Chapter 4 discusses the use of overlapping communication and computation in the parallel Diffpack library.

1.2.2 Improving data distribution

Uneven balancing of numerical work across the processors in a cluster will also lead to more time waiting, as the processors cannot continue before they have all completed their computations. We attempt to improve the balance of work across the processors, by introducing node-based partitioning to Diffpack in Chapter 5. Quality of graph-partitioning has a large impact on load-balancing, especially when using our implementation of node-based partitioning, and a brief comparison of how some of the functions in the graph-partitioning libraries METIS[5] and ParMETIS[6] perform is presented in Chapter 5. The results of our changes to the partitioner in Diffpack are presented in Chapter 6.

Chapter 2

Trace Libraries

In order to improve any parallel application we need reliable and accurate information about how and where the application spends its time and resources. There is need for a framework that can collect this information, and present the results and possible bottlenecks to the user. A good presentation helps the analysis, and makes it easier to decide what can be done to improve program performance.

To quantify and verify the impact of communication we have used several profiling packages, as well as the built-in timing support in MPI. There are several programs that offer benchmarking of MPI implementations and library code, i.e. MPIBench [7], SKaMPI [8] and MPPTest [9], but there does not seem to be that many software packages for performance analysis of user code. We have tested five different libraries that could be able to do the work that was needed in order to get a good impression of how our code was run and where performance bottlenecks were located.

When timing parallel software for development and optimization, accuracy and impact are important factors. We have tested several different packages that claim to profile and trace MPI programs with low impact on runtime, and our results are presented in this chapter.

2.1 Profiling packages

2.1.1 Intel Trace Collector

The Intel Trace Collector/Analyser (ITC/ITA) [10] is a combination of a library for tracing (Intel Trace Collector) and a GUI tool for trace analysis (Intel Trace Analyzer). It was formerly co-developed with VAMPIR, and developed by Pallas [11]. The collector (ITC) is a library that is included at compile time, and generates traces realtime. Calls to functions in the ITC library are added around the sections of user-code that are to be profiled, and details are then provided as parts of the traces that are produced. The ITC supports several output formats, where the format used by Vampir is one of the options. This format is easily readable, but there does not seem to be any tools for analysing the textual output, so it is not well suited for scripted analysis. The Single Trace Format is accepted for input to the Intel Trace Analyzer, and is a very good

solution for in-depth per-run analysis. However, it is not convenient for large numbers of runs, due to the time taken to investigate all the data, hence not suited for automatic analysis. We used the ITC/ITA suite version 6, although version 7 is now available.

2.1.2 Kojak

Kojak [12] is a comprehensive suite of tools and a GUI, that provides analysis functionality. The tracing part of kojak, EPILOG, uses PAPI [13] and spawns multiple temporary files to keep the counters on each separate during run. Just as with the ITC, calls to the EPILOG library must be added at specific places in the code to specify which sections to trace. Finally these distributed trace-files are merged to create one single output stream at the end. This merged output is stored in a special binary format, where tools provided can read the files and display the information in a more readable format. Several programs are supplied to read the information contained in the EPILOG (ELG) trace files. For example the program `elg_print` will convert the information in the ELG-files to ASCII, but making any accurate calculations from this textual output is difficult as the file format is not well documented. One solution is to open the non-textual output in a GUI visualiser, such as the analyser in the Kojak suite; CUBE [14]. We attempted this solution, but the software did either not compile (Version 2.1.1) or was not stable (Version 2.2b3). This might be more successful with more work.

2.1.3 mpiP

mpiP [15] is a small lightweight profiling library where statistics and output of the profiled code sections are output directly to text, enabling use of text-matching for analysis. It proved very helpful when doing comparison and data plotting, while programs such as the ITA or CUBE are better for understanding how MPI is performing and why. It provides large amount of information, and can well be used to analyze bottlenecks etc, but not as easily as with a graphical user interface. To profile using mpiP you do not have to add any code, as there is only one level of detail, but adding directives to turn tracing on and off are useful for limiting the size of trace-files. For all the other tools we tested code was added to define sections of code as entities, making it possible to identify the individual sections i.e. when using a analysis tool in a graphical user interface. Toolgear [16] can parse the output from mpiP and merge it with the source-code that was compiled into the traced binary, but for our use it is easier to get the information straight from the output.

2.1.4 TAU

TAU (Tuning and Analysis Utilities) [17] works much like the Intel Trace Collector and Kojak, by that you add calls to the profiling library at entry and exit of the particular piece(s) of code you would like to profile. The output from TAU can both be visualised in ParaProf and CUBE, but a lack of the analysis functions we require in ParaProf made it easier to use other packages to generate and analyse our traces.

2.1.5 VAMPIR

VAMPIR [18] shares many aspects with the ITAC, as they were part of the same package for some time. It works much the same way, and much of the same functionality is provided. It outputs to the OTF (Open Trace Format) which is a plus, and does not require a license to perform the traces. To run the Vampir Analyzer however, a license is required. Another plus with the support for the OTF is that other tools are starting to adopt this format, thus several tracing libraries can be visualized using the same package.

2.2 Using the Packages

We tested the different packages on a two-dimensional wave solver written in C. We built the program with the 5 different libraries in Table 2.1 plus the built-in `MPI_Wtime()` from the MPI library, both for blocking and nonblocking communication using the Makefile and FLAGS as shown in Listing A.2. Since the machines in our test-cluster *chilopodus* are all dual socket (Table 3.1), we used the *ppn* option of qsub to reserve both CPUs in a machine, and use a modified machine file to make sure that we only used one CPU per machine at any one time (Figure A.1). This enabled us to add multiple jobs at the same time, while avoiding problems with several processes sharing memory, CPU time and interconnect bandwidth. For all experiments in this report we repeat each combination of parameters and data at least five times, to minimise the variations a shared system will introduce from one run to another. We use the median of the five runs to be sure we report reliable times. Since the processors report different times in the profiled part of the code, an average is used where this is necessary to get an impression of the overall time spent in these sections. The syntax for the different libraries is presented in Table 2.1.

We successfully managed to use both mpiP, ITC and TAU to generate trace data that we could then analyse. We were also able to generate traces using KOJAK and VAMPIR, but ran into problems when attempting to analyse the output. All the different profiling libraries produce similar results despite differences in size, and they all have a microsecond resolution, apart from mpiP which outputs only at the millisecond scale. We found the combination of a high-resolution tool with a GUI display (ITA) and a different tool (mpiP) to verify the results to be a reliable approach. And as we see from Figure 2.1 they report similar times for our two-dimensional wave solver. This confirms our confidence in the accuracy of the libraries. As you can never use more than one library at the same time, small differences in times can be reported by the libraries simply because it is not produced at the same time. We will use the ITA to analyse output from ITC, and mpiP to confirm our results for the remainder of our experiments.

2.3 Using the Intel Trace Analyser and mpiP

The Intel Trace Analyser is a GUI tool available on both Windows and Linux to analyse the traces generated by the Intel Trace Collector. It has certain requirements to the libraries available to it, such as which QT versions it will run

VAMPIR

Init	#include <vt_user.h>
Entry	VT_TRACER(”my_section”);
Exit	N/A

mpiP

Init	MPI_Pcontrol(0);
Entry	MPI_Pcontrol(1);
Exit	MPI_Pcontrol(0);

ITAC

Init	VT_symdef(123, ”my_section”, ”USER”); VT_traceoff();
Entry	VT_traceon(); VT_begin(123);
Exit	VT_traceoff(); VT_end(123);

KOJAK

Init	#pragma pomp inst init;
Entry	#pragma pomp inst begin(my_section);
Exit	#pragma pomp inst end(my_section);

TAU

Init	#include <Profile/Profiler.h>
	TAU_PROFILE_TIMER(my_section, ”iteration”, ”No arguments”, TAU_DEFAULT); TAU_PROFILE_SET_NODE(0);
Entry	TAU_PROFILE_START(my_section);
Exit	TAU_PROFILE_STOP(my_section);

Table 2.1: Code added to C-code in order to call the required tracing libraries at both entry and exit of traced sections of code. At initialisation libraries are included and tracing is turned off if possible.

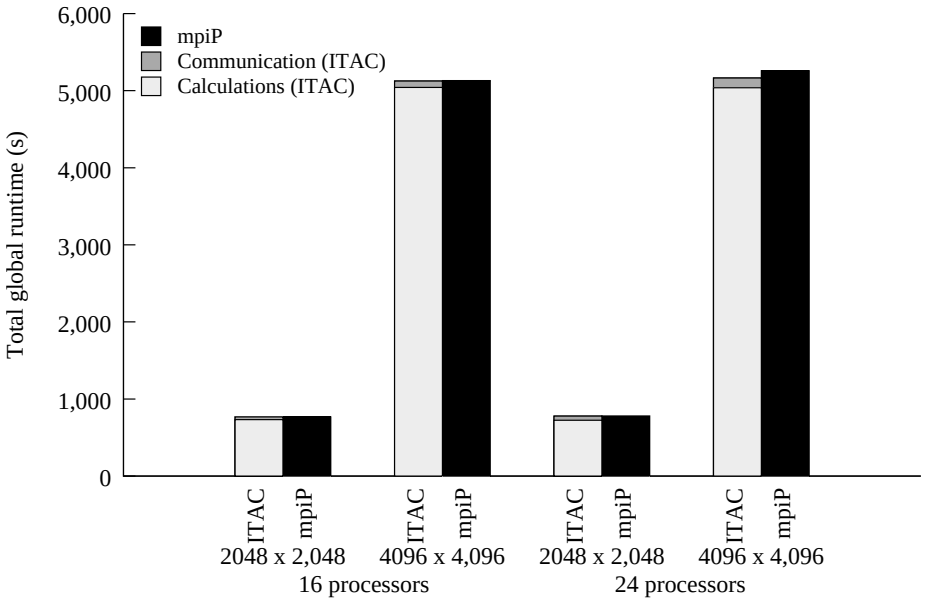


Figure 2.1: Global times reported for the 2D wave solver by ITAC and mpiP. ITAC was used to divide the elapsed global time into communication and computation, while mpiP was used to check the output from ITAC. The difference when using 24 processors on a 4096×4096 matrix is because the libraries cannot be used concurrently, and the times reported are from different executions of the same program.

on. We ran the ITA under X11 on a regular Linux desktop machine (Gentoo with QT version 3.3.4). Initially the ITA presents the user with a timeline of the data loaded, but many graphs and tables can be added from several menus in order to fully understand how the programme that was traced behaved. I found the Chart Event Timeline (Charts ⇒ Event Timeline) very useful (Figure 4.2). You can zoom in and out on the timeline, looking only at the portion of the timeline that is interesting for the user. Under Advanced Options you can also choose to filter the data according to several criteria, i.e. which processors are involved, the communication density, which type of communication that is in use etc.

As mpiP presents the information in text-files, only a few script were required to gather the information we required to be able to use it.

Chapter 3

Testing overlapping of communication and computation

3.1 Introduction

Sub-optimal design and implementation of inter-processor communication and computation can lead to penalties in performance and increase the resource-consumption considerably. As the goal of our work is to improve inter sub-domain communication for typical numerical methods for solving partial differential equations, we chose first to implement an standard explicit wave solver that has similar communication characteristics. Explicit wave solvers require only simple computations at each time-step compared to the computations at each iteration for example in the conjugate gradient method, thus changes in the communication and communication pattern will be emphasised.

3.2 Wave solvers

Overlapping communication and computation is central for parallel scientific computing [19], since it enables computers to continue with local work while communication is still in progress. We have implemented two- and three-dimensional explicit wave solvers that can overlap communication with computation, by using non-blocking communication calls in MPI. This overlapping of communication and computation can be turned on and off, in order to see how much this difference affects the overall program progress. The non-blocking functions `MPI_Isend()` and `MPI_Isecv()` are used when overlapping communication and computation, and the blocking functions `MPI_Send()` and `MPI_Recv()` are used otherwise. As parallel wave solvers must exchange boundary-values at every time-step, communication will have significant impact on the program and its overall execution. This way we hope to investigate how much these different communication approaches differ, and how it affects program flow.

Standard explicit wave solvers are used to find the state of a wave at points in time after a given initial state. The state of the wave is computed for points in time spaced by time-steps, and it is based on the state of the wave in the previous time-step. So for a solver to find the state at a certain time, it must start at the initial state and loop through all time-steps until it reaches the desired time-step. For parallel wave solvers inter-sub-domain communication must occur at each time-step, illustrated in Listing 3.1.

Whether or not a system can actually provide an advantage when overlapping communication with computation depends on the underlying MPI implementation and hardware. Our two- and three-dimensional wave solvers were programmed and traced both with overlapping communication and computation turned on and off, to see how this would affect elapsed wall-time.

	chilopodus	**tre**
CPU	Dual Socket Itanium2	32×Power 4 64-bit
Core Frequency	1300 MHz	1300 MHz
Cache	16 KB L1, 256 KB L2 3 MB L3	128 MB L3
System Memory	4 GB per	192 GB
Operating System	Debian GNU/Linux 3.1	AIX5L 5.1
Kernel	Linux 2.6.8 ia64 (SMP)	
Queue system	Torque 1.2.0 (p0)	LoadLeveler/ Maui
MPI Implementation	MPICH 1.2.6	
Interconnect	Gigabit Ethernet	Shared Memory

Table 3.1: Computer systems referred to in this document. *chilopodus* is a cluster of dual-socket Itanium2 machines with ethernet interconnect, while *tre* is shared-memory machine with 32 CPUs

3.3 Implementation

Wave solvers are in nature easy to divide into smaller sub-domains because of their uniform shape, making it easy to solve the partial differential equation on parallel systems. A simple two-dimensional grid with division between the boundary points that need to be exchanged with neighbours, and the interior points each processor must keep locally for is illustrated in Figure 3.3 (4 processors).

Listing 3.1: Pseudocode for wave solver

```
generateInitialCondition();
exchangeInitialCondition();

while(iterations) {

        updateEdgepoints();

        if(BLOCKING) {
                if(DIMENSIONS > 0) {
                        sendEdgesWest();        recvEdgesEast();
                        sendEdgesEast();        recvEdgesWest();
                }
                if(DIMENSIONS > 1) {
                        sendEdgesNorth();       recvEdgesSouth();
                        sendEdgesSouth();       recvEdgesNorth();
                }
                if(DIMENSIONS > 2) {
                        sendEdgesUp();          recvEdgesDown();
                        sendEdgesDown();        recvEdgesUp();
                }
        }
        else {
                if(DIMENSIONS > 0) {
                        iSendEdgesWest();       iRecvEdgesEast();
                        iSendEdgesEast();       iRecvEdgesWest();
                }
                if(DIMENSIONS > 1) {
                        iSendEdgesNorth();      iRecvEdgesSouth();
                        iSendEdgesSouth();      iRecvEdgesNorth();
                }
                if(DIMENSIONS > 2) {
                        iSendEdgesUp();         iRecvEdgesDown();
                        iSendEdgesDown();       iRecvEdgesUp();
                }
        }

        updateInteriorPoints();

        if(!BLOCKING) {
                waitAll();
        }
}
```

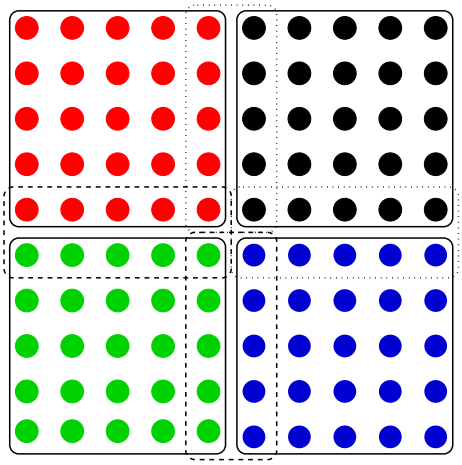


Figure 3.1: A 10x10 grid partitioned into 4 sub-domains. Different colours illustrate different sub-domains, while dotted lines illustrate borders where we have to exchange values between each time step

We can calculate our boundary points (points within dotted areas) initially, then start exchanging these with the correct neighbours using the nonblocking communication calls `MPI_Isend` and `MPI_Irecv`, in this example five points with each neighbour. While we send these five points to each of our neighbours and receive the five points they have sent us, we can compute the rest of our data (points outside dotted areas, 16 points per processor in this example). When we return from computing our internal points, we wait for the exchanges to finish by calling `MPI_Wait` and continue to the next time step. This way overlapping communication and computation can hide the time it takes the data to traverse the interconnect, although latency due to set-up and tear-down of communication is unavoidable.

The different versions of our wave solvers (Pseudo code in Listing 3.1) were tested on *chilopodus* (Table 3.1), the test-cluster at Simula.

To get an impression of what overlapping communication and computation can do on this type of computer system we used the mpiP library to see how the workload was divided between MPI library routines and computational time finding the solution. As mpiP is well suited for scripted data analysis of large numbers of runs, we could produce the graphs presented in Figures 3.2(a) and 3.2(b), while the ITA could then be used to produce higher detail information using the GUI analyzer should this be necessary.

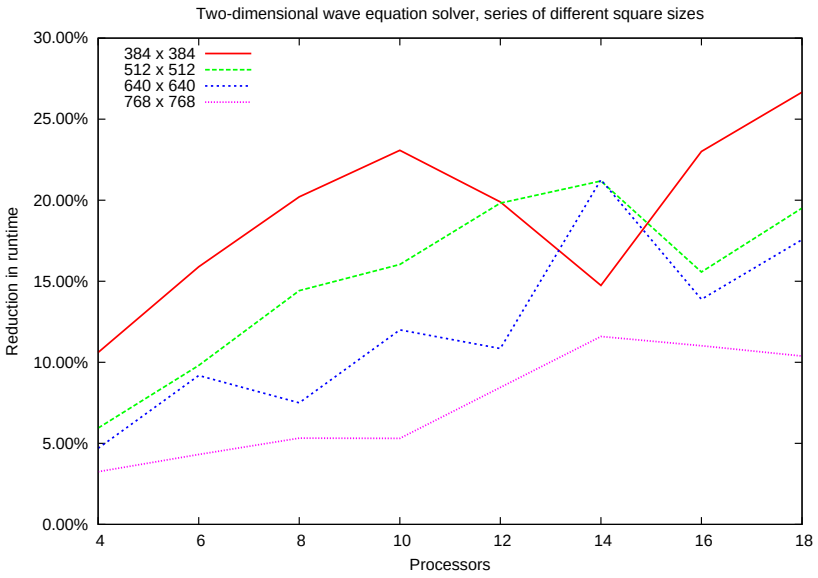
3.4 Results

Although the data from the two-dimensional wave solver varied extremely from one run to another (despite the use of medians [n=5]) it still shows an overall trend indicating that an increased number of processors used to solve the equation, will lead to an increased performance gain by overlapping communication and computations. Increasing the number of time steps did help to a certain degree, but not enough to get the same results from one trace to the next. This is most likely due to the fact that communication is low compared to the memory requirements; Imagine a local 32x32 2D-matrix, with a total of 1024 points. This would require that 32 points be exchanged from one processor to another per iteration. Whilst for a local 10x10x10 3D-cube, requires that 100 points be exchanged from one processor to its neighbour per time step. As expected the 3D wave solver provided more stable results, and confirmed all the observed trends from the 2D solver;

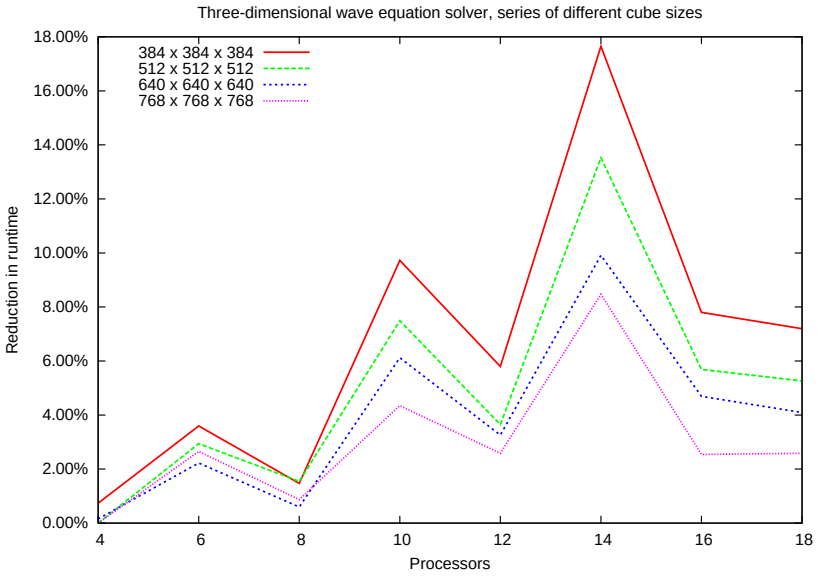
1. Overlapping communication with computation performs better when using more processors

2. Overlapping communication with computation performs better for smaller data-sets

As we added one dimension to our problem, both the time to compute all points and exchange them takes longer, and makes it easier to generate reliable test results. So despite the fact that we could not make any conclusion by the two-dimensional wave solver alone, with three dimensions the generated timings are stable. They confirm the theory and show that overlapping can indeed lead to a reduction of time needed per time-step.



(a) 2D Wave solver, 5000 time steps



(b) 3D Wave solver, 1000 time steps

Figure 3.2: Reduction in runtime by overlapping communication and computation. 4 different problem sizes

Overall it appears that smaller grids will benefit more from overlapping communication and computation, as they will have a higher communication to calculations ratio. Using 16 processors is not uncommon, so we chose to use 16 processors to see how changes in the amount of communication and calculations affects the performance of the solver. Table 3.2 shows how the relative amount of communication increases at a slower rate than the computations when the problem size grows, hence overlapping this communication with the computations reduces the runtime less as the problem grows. Computational points are the total number of points that are computed at each processor, and worst-case communication refer to the amount of data the processor with highest surface area towards its neighbours must exchange. The 384x384x384 cube in Table 3.2 is sketched in Figure 3.4. Dark grey partitions have more data to exchange with neighbours, and all other processors must wait until processors allocated dark grey areas are finished.

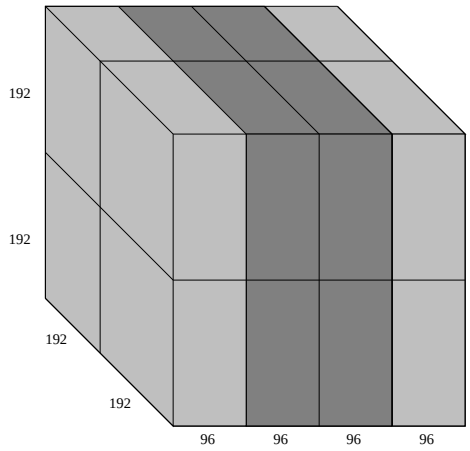


Figure 3.3: A 384^3 cube partitioned over 16 sub-domains

In general using more of the processors in the cluster lead to a larger reduction in runtime when overlapping communication and computations, as the runtime for the blocked implementation increases. This is again because a processor in the blocking code must communicate with all its neighbours sequentially, so as the number of neighbours increases so does communication. For unblocking communication, exchanges with all neighbours can be done while the interior points are computed, thus saving more and more time as the number of processors increase.

From the 3D solver there is a clear trend indicating that using a number of processors that are the product of two prime numbers, and therefore not divisible into three dimension (i.e. 6, 10 and 14 processors) have artificially large reductions in runtime. This is because the problem-cube does not divide easily into cube-like parts, but is divided into parts that have large differences between the length of the sides. Table 3.3 shows how this can cause the reduction of runtime to be so different for 12 and 14 processors. Solving the system with 14 processors results in a higher worst-case communication cost from one processor to another, as the processors have one side with length 384. This results in a larger surface area between the processors, and the communication

Size	X-side	Y-side	Z-side	Comp. points	Worst-case communication	Ratio
384^3	96	192	192	3538944	184320	5.21%
512^3	128	256	256	8388608	327680	3.91%
640^3	160	320	320	16384000	512000	3.13%
768^3	192	384	384	28311552	737280	2.60%

Table 3.2: Size of the three-dimensional cubes generated when we use 16 processors. The cubes are the same size for all processors for all global problem-size individually.

increases. Since the communication load per processor is higher for 14 processors, overlapping this communication with the computation leads to a larger reduction in runtime compared to using 12 processors. For 12 processors the worst-case communication between two processors is not only lower, but the amount of computation per processor is higher, contributing to this effect.

12 Processors			
Number of sub-domains	Size	Comp. points	Worst-case communication
12	128x192x192	4718592	196608
14 Processors			
Number of sub-domains	Size	Comp. points	Worst-case communication
12	54x192x384	3981312	336384
2	60x192x384	4423680	340992

Table 3.3: Communication and calculation associated with solving a 3D wave system with 12 and 14 processors. Using 12 processors results in equal cube-size for all processors, but using 14 processors results in 2 cubes that are larger than the other 12.

Chapter 4

Improving inter-subdomain communication

Domain decomposition methods with overlapping sub-domains result in duplication of data and communication characteristics similar to the parallel solver of wave equations in the previous chapter. This can increase the amount of communication that is required compared to methods where the sub-domains do not overlap, depending on the size of the overlap. As X. Cai mentiones in [20], the points allocated to each sub-domain can be divided into four categories as described in Table 4.1 and illustrated in Figure 4.1.

Point Type	Description	Colour
1. Interior non-overlapping	Belongs to this processor only, and must therefore be computed here	White
2. Computational Overlapping	Belongs to more than one processors, but are computed at the local processor	Light grey
3. Non-Computational Overlapping	Belongs to more than one processor, but are computed by another processor	Dark grey
4. Internal Boundary	Internal points on the overlap, are computed by another processor	Black

Table 4.1: Domain Decomposition point types

This categorisation of points enables us to apply the same principles as in our wave equation solvers; computing interior computational points while sending and receiving the previously computed overlapping computational points. Categories 1 and 2 are computed on the local processor, but only category 2 are exchanged with other processor. To apply the same approach of optimisation here, we compute the points in category 2, then start exchanging these with our neighbours in return for points in category 3 and 4. When we have finished computing our category 1 points, we finish the exchange and continue to the next iteration. If the amount of data to be exchanged is large

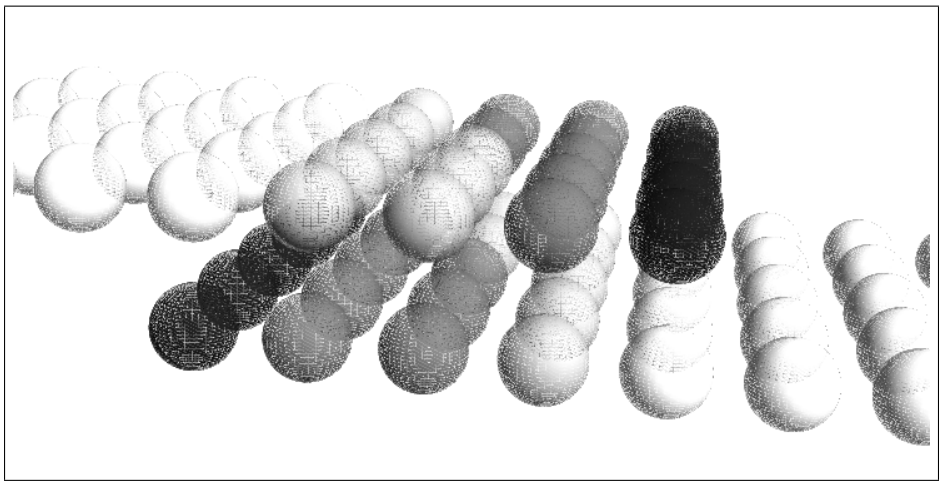


Figure 4.1: Two overlapping sub-domains at an internal boundary, thhe overlapping section is 5×5 points. Colour-coding is explained in Table 4.1

enough, this overlapping of communication and computation, or masking of network latency behind the computation of interior points will result in noticeable improvement.

4.1 Implementation

We created the functions `startExchangeValues (VecSimple(real)& vec)` and `finishExchangeValues (VecSimple(real)& vec)` (Listings 4.3 and 4.4 to break up the communication as required. `startExchangeValues()` returns immediately, and allows the program to carry on doing computations while `finishExchangeValues()` returns when the communication is finished. The matrix-vector product exchanges the overlapping points by calling `startExchangeValues()` as soon as the overlapping points are computed, and calls `finishExchangeValues()` when it is finished computing the interior points. The new version of `matvec(...)` is described in Listing 4.5. The non-blocking send function `MPI_Isend()` is used whether we overlap communication and computation or not, to avoid deadlocks.

The differences between overlapping communication and computation and not are highlighted in Listings 4.1 and 4.5.

Listing 4.1: Non-overlapping communication

```
for i in computational points {
        rstart = ad.irow(i);
        rstop  = ad.irow(i+1);
        for (r = rstart; r < rstop; r++)
                y(i) += A(r) * x(ad.jcol(r));
}

exchangeValues( computational overlapping points in y );
```

Listing 4.2: exchangeValues()

```
for n in neighbours {
        MPI_Isend( computational overlapping points with n, to n );
}

for n in neighbours {
        MPI_Recv( non-computational overlapping points with n
                        and internal boundary points at n, from n );
}
```

Listing 4.3: startExchangeValues()

```
for n in neighbours {
        MPI_Isend( computational overlapping points with n, to n );
}

for n in neighbours {
        MPI_Irecv( non-computational overlapping points with n
                        and internal boundary points at n, from n );
}
```

Listing 4.4: finishExchangeValues()

```
for n in neighbours {
        MPI_Wait( for complete reception of data from n );
        MPI_Wait( for complete send of data to n );
}
```

Listing 4.5: Overlapping communication and computation

```
for i in computational overlapping points {
        rstart = ad.irow(i);
        rstop  = ad.irow(i+1);
        for (r = rstart; r < rstop; r++)
                y(i) += A(r) * x(ad.jcol(r));
}

startExchangeValues( computational overlapping points in y );

for i in computational non-overlapping points{
        rstart = ad.irow(i);
        rstop  = ad.irow(i+1);
        for (r = rstart; r < rstop; r++)
                y(i) += A(r) * x(ad.jcol(r));
}

finishExchangeValues( computational overlapping points in y );
```

We investigated the traces that were produced at both the level of a single matrix-vector product, but also as a set of several iterations. We defined several tracing segments to show how time was divided between the different interesting sections in the code;

- section where overlapping computational points are computed
- section where non-overlapping points are computed
- the MPI library

4.2 Results

4.2.1 Data sets

We solved a Poisson problem using the conjugate gradient method to see how Diffpack was affected by changes in code and problem properties. We have used two test-grids, both unstructured grids with properties described in Table 4.2. We will refer to these two as the small and the large grid respectively.

	Smaller Data-set	**Larger Data-set**
Name	heart-muscle.grid	fine_refined-fixed.grid
Nodes	11306	28283
Elements	56568	162120
Element type	**`ElmT4n3D`**	**`ElmT4n3D`**

Table 4.2: Grid properties

4.2.2 Chilopodus

We installed the Intel Trace Collector on the test system *chilopodus*, and tested our new version of the matrix-vector product which overlaps communication and computation against the original code.

At first the difference between overlapping communication and calculation (Figure 4.4) versus not overlapping (Figure 4.2) is not obvious; The total time to solve for one complete solution of our Poisson problem has not changed. We turn to the ITA [10] which reveals that although the runtime is the same, the structure of the communication and computations has changed as expected. For a single matrix-vector product the actual time spent in the MPI library when overlapping communication and computation has decreased by about 50% compared to the version where overlapping of communication and computations was not used. (Table 4.4) This seems promising, but why does this not result in a reduced runtime? As the solution of a system of linear equations also uses the inner product, collective communication is used to find the result of this. (Implemented using `MPI_Allreduce` in Diffpack). The solve function calls `MPI_Allreduce` three times after it returns from the matrix-vector product (`MPI_Isend/MPI_Irecv` combination) in one iteration. All these calls to `MPI_Allreduce` are the single largest contribution to total MPI-time both when overlapping communication/computation and when not. Looking at the data distribution in Table 4.3 it seems obvious that the processors will not take the same amount of time to finish computing the overlapping points; The workload actually differs by around 10%. (Maximum of 42001 and minimum of 37663 computational points). This means that all processors will have to wait for the processor with the heaviest load before they are able to continue, both after the matrix-vector product and the inner product.

The result is that time spent is shifted from `MPI_Send/MPI_Recv` calls to `MPI_Allreduce`, without actually reducing time spent in MPI. The output from the ITA confirms this, and it seems that there is little advantage overlapping communication and computation as the processors still will have to wait for each other. This shows that blocking communication also served a function

Partition	Computational points		Total
	Non-overlapping	Overlapping	
1	33884	4723	38607
2	37394	2979	40373
3	34103	5180	39283
4	34312	3820	38132
5	35425	4421	39846
6	37321	2811	40132
7	37542	3313	40855
8	36631	3160	39791
9	34204	5177	39381
10	33653	4010	37663
11	34570	4887	39457
12	36525	2502	39027
13	32496	6020	38516
14	36355	3389	39744
15	34907	4717	39624
16	39338	2663	42001

Table 4.3: Computational points per sub-domain on 16 processors

	Blocking communication	*Non-blocking communication*
Matrix-Vector Product		
MPI_Isend	2.161 s	1.403 s
MPI_Recv	5.124 s	N/A
MPI_Irecv	N/A	0.186 s
MPI_Wait	0.081 s	2.251 s
Total	7.366 s	3.840 s
Inner Products		
MPI_Allreduce	10.432 s	13.750 s
Total for 519 conjugate gradient iterations		
MPI Time	17.798 s	17.603 s
User Code (For Comparison)	163.674 s	163.919 s

Table 4.4: Breakdown of time for 519 conjugate gradient iterations on small data-set. Timings reported by the ITA. (Two levels of refinements, defined in Section 5.3)

as synchronisation, and that much of the reported MPI-time was spent waiting rather than receiving data. All this waiting, either as time in `MPI_Recv` or `MP_Allreduce`, increases the time spent on communication. Hence fixing the uneven distribution of work between the processors could decrease the total communication time, and hopefully also reduce global runtime. As MPI-time does not dominate the total time spent solving our test-sets, it is only important that the total time spent on computation is distributed evenly. Thus computation is finished as closely as possible in time, and also the crucial finishing call to `MPI_Wait`. If communication is larger, and it approaches the time it takes to compute the interior points (i.e. with large sections of the sub-domains overlapping at the interior boundaries) it may also be necessary to balance both the internal and the boundary points individually. This was never an issue when not overlapping; it was only important that the total amount of points (Interior points and overlapping boundary points) were balanced so that the finishing `MPI_Send()` and `MPI_Recv()` could be called as closely in time as possible. Having said that, the ratio of communication and computation seen in our test sets are fairly typical for domain decomposition methods, so we can assume that communication never will take longer than the time to compute the interior points.

Figure 4.2: One conjugate gradient iteration where communication is not overlapped with communication in the matrix-vector product, 16 sub-domains (Small data-set with two pre-refinements)

We have attempted to test with typical sets of data to get a an impression of the real ratio between communication and calculation for conjugate gradient methods on unstructured mesh workloads. The time taken to exchange the data is relatively small compared to the time spent doing computations, as we can see from Figures 4.2 and 4.3 (Matrix-vector products are shown in turquoise, inner products in yellow, communication in red with black lines, and collective communication in red with blue lines. Purple shows code in the `solve()` call, inside `LinEqSolver`. The quantitative timeline shows little time dominated by MPI, so we should not expect overlapping communication

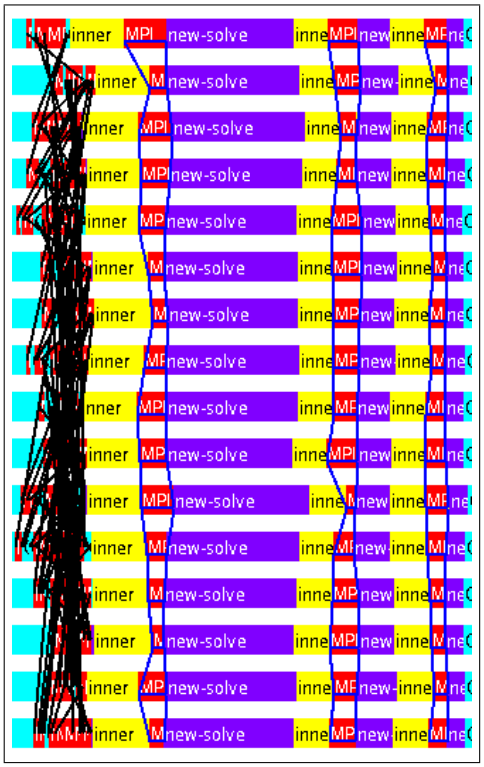


Figure 4.3: Enlarged view of the finishing communication and inner products for one conjugate gradient iteration, 16 sub-domains (Small data-set with two pre-refinements)

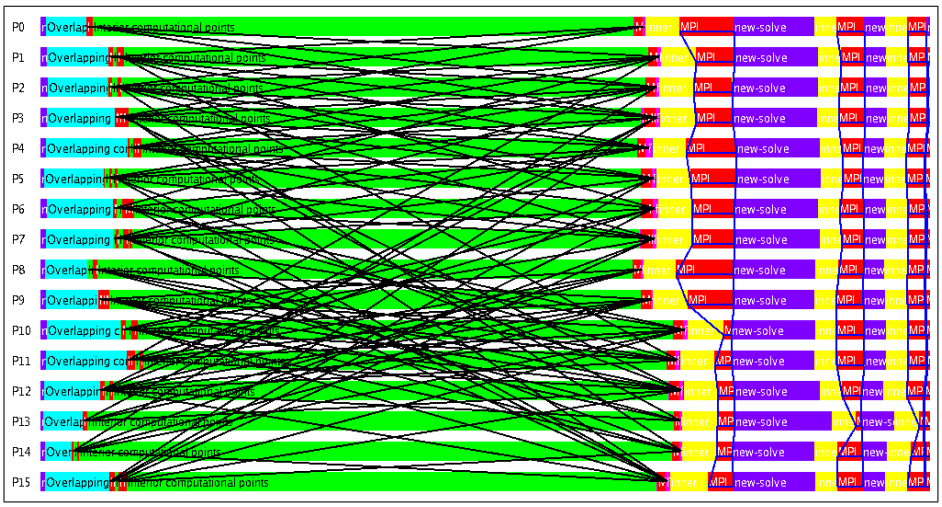


Figure 4.4: One conjugate gradient iteration where communication and computation are overlapped in the matrix-vector product, 16 sub-domains (Small data-set with two pre-refinements)

and computation to reduce the overall runtime largely. For parallel programs where we need to exchange larger amount of points between iterations, overlapping communication and computation could have a larger impact. Looking at our test data, the largest number of points to be communicated between two processors is somewhere around 4000 points. The time it takes 4000 points of 8 bytes to cross a gigabit interconnect is around:

$$\frac{64 \times 4000 bits}{1.0 \times 10^9 bits/s} = 0.000256s \tag{4.1}$$

The effective data-rate of the test system *chilopodus* was measured to be around 110 MB/s. One iteration takes around 2.2×10^{-2} seconds (acquired from the ITAC), so overlapping the 2.6×10^{-4} seconds could really only result in a reduction of:

$$\frac{2.6 \times 10^{-4}s}{2.6 \times 10^{-4}s + 2.2 \times 10^{-2}s} = 1.17\% \tag{4.2}$$

A one percent reduction in time seems reasonable looking at the reduction in MPI time in Figure 4.4 (Around 1.1% reduction here). For the time to complete 519 conjugate gradient iterations, we see how there is a small reduction in MPI-time overlapping communication and computation. These numbers will change somewhat between runs, but they clearly indicate that the time saved on overlapping communication and computation is minimal. The variation in CPU time spent computing points varies more in our example than the theoretical reduction in MPI-time due to overlapping communication and computation. Due to the uneven distribution of the workload the blocking communication also served a synchronising function, and removing this resulted in the shifting of time from the `MPI_Send/MPI_Recv`-functions to the collective operations.

The results we have presented here are for 16 processors, and reducing the number of processors should reduce the effect of overlapping communication with computation as there is less communication and more work per processor. The speedup for both overlapping communication and computation is presented in Figure 4.5. As we have measured speedup to be close to linear, it is clear that the effect of communication is minimal.

4.2.3 Tre

The same combinations of communication-type and data-set tested on *chilopodus* were also tested on *tre*. Figure 4.6 illustrates the speedup we achieved.

4.3 Conclusion

From our results we see that overlapping communication and computation as implemented here and tested on *chilopodus* and *tre* will not lead to better

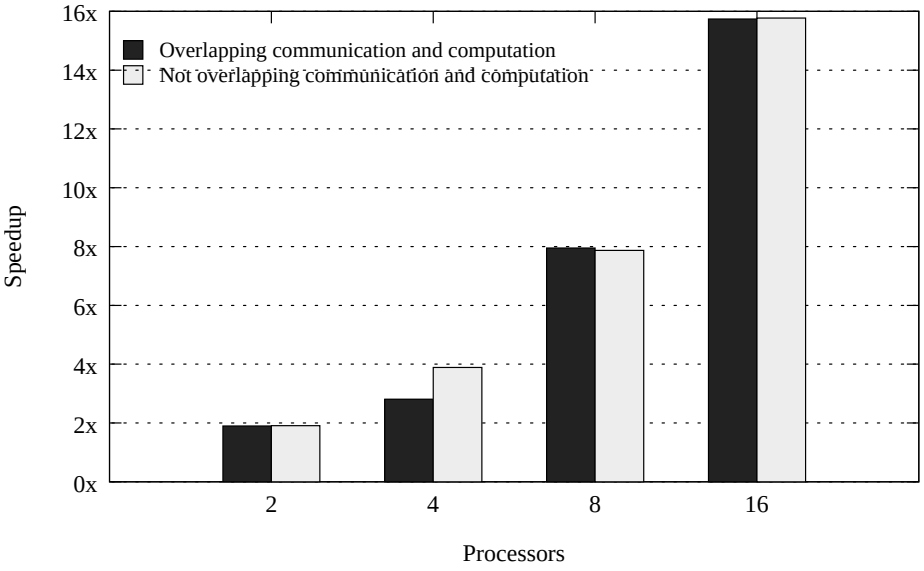


Figure 4.5: Speedup for 500 conjugate gradient iterations on *chilopodus*. Small data-set, two levels of pre-refinement (Explained in Section 5.3). Non-linear speedup for overlapping communication and computation on four processors is due to an unknown effect disturbing the MPI-communication

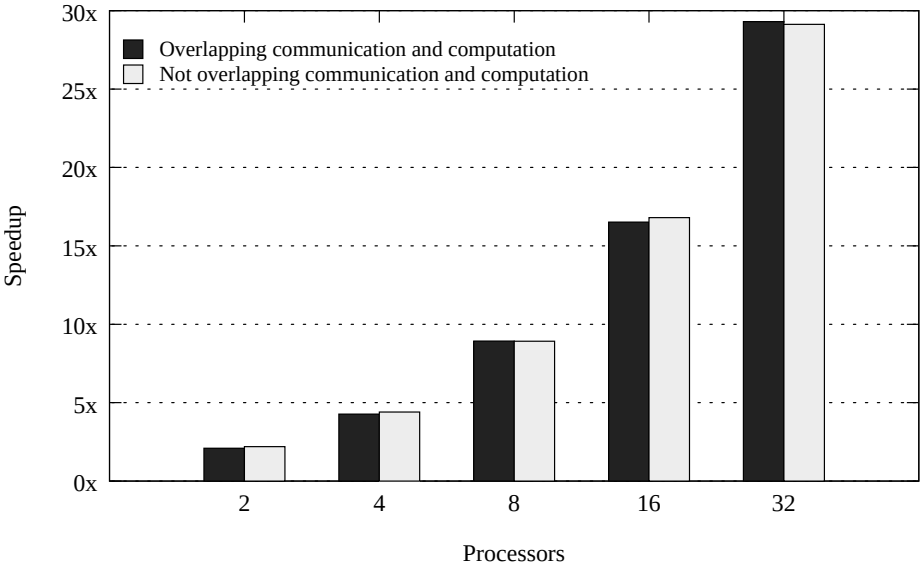


Figure 4.6: Speedup for 512 conjugate gradient iterations on *tre*. Small data-set, two levels of pre-refinement (Explained in Section 5.3).

performance with the type of communication that we have tested. The amount of communication for the type of numerical methods we have tested is such that overlapping it with computation can be measured reliably. Through this section we discovered that balancing the workload more evenly across the subdomains should show some performance friendly characteristics. A more even distribution of computational points would decrease time spent on waiting for the processor with most work, thus reduce runtime whether overlapping of communication and communication is used or not.

Chapter 5

Improving Data Distribution

In Chapter 4 we saw that the total number of computational points varied varied from one sub-domain to another. Partitioning our smaller test-grid into 16 sub-domains for example, resulted in a difference at around 10% from the sub-domain with the most to the sub-domain with the least points to compute. If we can reduce this difference, and in turn reduce the time spent waiting for the processor with the most computational points, we can hope for a reduction of the overall runtime of the matrix-vector product and the inner product.

A distribution of the overlapping points could be implemented such that it could serve as a method to balance the computational load. This approach would need to assume that there are enough overlapping points available to completely adjust the differences in computational points from one sub-domain to another. This assumption is not always possible, i.e. if a sub-domain has a relatively low ratio of overlapping- versus computational-points, for example when using first order overlaps. Diffpack can redistribute overlapping points using `distrbuteOverlapPts()` when the overlaps are so large that it is possible for overlapping points to shift from one sub-domain to another.

Computational work is effectively the amount of computational points as described in Chapter 4. So another approach to this problem is to make the distribution of the computational points as even as possible during partitioning, not after.

To easily assess the evenness of the partitions that are created we define an *imbalance factor* which we will use throughout this report; The size of the largest partition divided on the average size of the partitions. A large imbalance factor signals a maximum far from the average thus poor balance between the partitions. The maximum is the deciding factor for progress, as no processor can continue until the processor with the largest partition has finished. With partitions approaching equal sizes, the maximum approaches the average and the imbalance factor approaches one.

5.1 Mesh partition

An even partitioning of the data is crucial for achieving good performance on parallel computers. Diffpack can use both METIS [5] and ParMETIS [6] to partition the mesh among the processors, which one is decided by the user. Both

METIS and ParMETIS (parallel version of METIS) partition graphs with good results as we shall see.

5.1.1 METIS and ParMETIS performance

The method `ParMETIS_V3_PartKway()` can be used to perform a parallel k-way partitioning of graph, and is used in Diffpack to partition graphs where graph-nodes represent elements in the mesh that is provided by the user. `ParMETIS_V3_PartKway()` takes a parameter *imbalance tolerance*; A higher imbalance tolerance[1] results in a lower number of edges cut during partitioning (edgecut), but opens for a more uneven partitioning. The opposite will lead to a more even partitioning at the cost of a higher edgecut.[2] Since generating even partitions of the mesh is our primary focus, and the fact that we will overlap communication and computations, a higher edgecut is not a major concern.

We tested `ParMETIS_V3_PartKway()` with a range of imbalance tolerances to check how this affected the overall partition sizes and the edgecut. Since we are only looking for the best possible partitioning at this stage, a serial implementation such as METIS and not only ParMETIS is also interesting, despite the fact that this is a serial algorithm running on a parallel system. Of course for production workloads parallel partitioning is preferred as the work is distributed and the contribution towards runtime of the program itself can be minimised.

When it comes to METIS, the manual [21] says that `METIS_PartGraphKWay()` "should be used to partition a graph into a small number of partitions (less than 8)" [21] p. 21. But as mentioned, our major concern it to produce the best partitions, and we found `METIS_PartGraphRecursive()` to do so. Table 5.1 shows partitions generated from the same input-graph by different functions in METIS and ParMETIS.

As expected a lower imbalance tolerance generates more even partitions, but a higher edgecut. Interestingly an imbalance tolerance of 1.02 results in a lower edgecut than 1.05, and also more even partitions at the same time. `METIS_PartGraphRecursive()` performs the best, with perfectly even partitions. We chose to use this function to partition our grids from now on, as it clearly generates the best partitions with respect to the balancing of graph-nodes. It is also a stable candidate as it does not require the user to provide any parameters, only the desired number of partitions to split the graph into.

5.2 Improving mesh distribution

5.2.1 Current approach

Each node in a mesh represents a point that needs to be computed, either by the local processor or another in the cluster. Traditionally Diffpack partitions the input-mesh by generating a graph representing the elements, achieving partitions with even amounts of elements. (Figure 5.2, Left)

If an overlap of elements in the neighbouring sub-domains is required (The elements are allocated to two or more of the sub-domains) the traditional element-

[1] Maximum is number of subdomains

[2] Minimum is 1.00. A value of 1.05 is recommended

Partition	METIS_PartGraph_		ParMETIS_PartGraphKWay() Imbalance tolerance			
	Recursive()	KWay()	1.00	1.01	1.02	1.05
1	28284	27902	28311	28315	28395	27614
2	28284	28954	28285	28273	28427	28367
3	28284	28971	28278	28238	27724	28635
4	28284	28237	28285	28278	28391	27619
5	28284	27477	28302	28310	28330	28432
6	28284	27800	28277	28421	28531	28659
7	28284	27804	28240	28296	28328	29000
8	28284	28952	28306	27986	27142	27334
9	28284	27653	28303	28275	28332	29003
10	28284	29126	28287	28272	28336	27307
11	28284	27473	28270	28363	28385	28338
12	28284	29130	28275	28226	28451	27147
13	28284	27703	28284	28265	28386	28605
14	28284	28079	28286	28455	28431	28247
15	28284	29054	28273	28291	28398	28702
16	28284	28229	28282	28280	28557	29535
Max	28284	29130	28311	28455	28557	29535
Average	28284	28284	28284	28284	28284	28284
Balance Ratio	1.00	1.03	1.00	1.01	1.01	1.04
Edgecut	14591	14365	23220	16116	15664	15716

Table 5.1: Size of sub-grids generated from the small grid by different METIS and ParMETIS functions

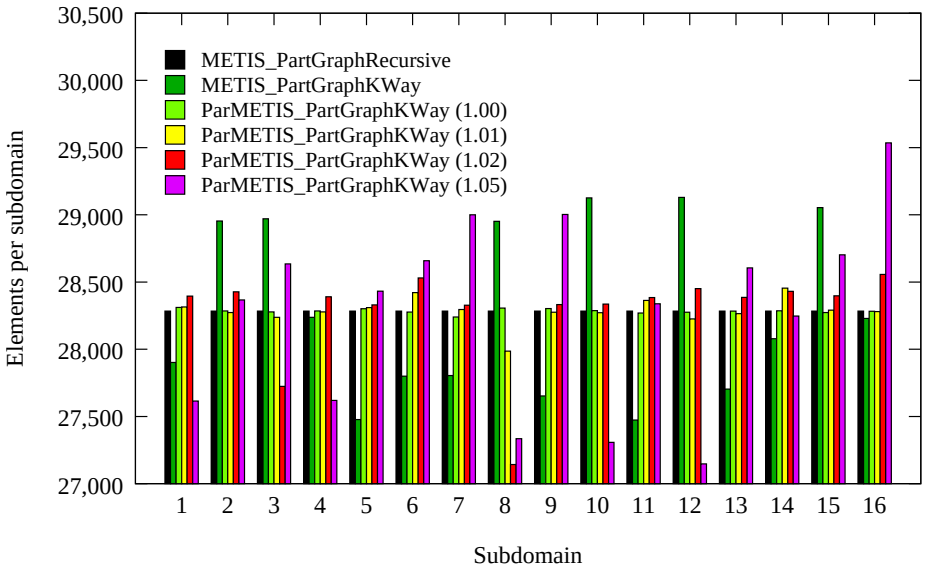


Figure 5.1: Size of sub-grids generated from the small grid by different METIS and ParMETIS functions

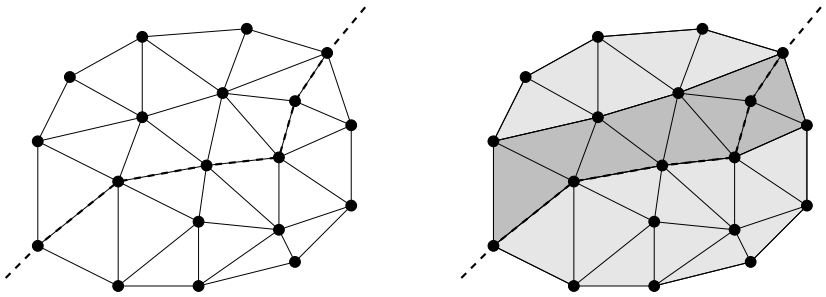


Figure 5.2: Element-based partitioning of unstructured 2D mesh. Figure to the left is the output from the partitioner, and Figure to left shows overlapping elements in dark grey, while the two sub-domains are shown in a lighter shade. Dotted line is the split generated by the partitioner

based approach must call functions to add overlaps after it has received the initial sub-domains from the partitioner. Adding overlaps is done by adding elements in one sub-domain to the neighbours sub-domain. Since all sub-domains already are of equal size this process must be careful not to disturb the balance of elements between sub-domains. The logic to add overlaps is implemented in several functions in Diffpack; `fixInternalBoundariesNode()` adds a first-order overlap, and `addOverlap()` is used if larger areas of overlaps are required. It is important to understand that the partitioner does not help in this process at all, as it only attempts to split the graph into partitions of equal size. Figure 5.2 shows an element-based partitioned mesh straight from the partitioner (Left) and how it might look with overlapping elements (Right).

5.2.2 Introducing node-based partitioning in Diffpack

Computational work is related to the number of nodes in a mesh, but for unstructured mesh an even distribution of elements does not necessarily imply an even distribution of nodes (computational points), as the number of nodes is not necessarily proportional to the number of elements in any given part of the mesh. A trivial example is two tetrahedrons sharing a point, or two tetrahedrons sharing a side in 3D. The number of elements is always two, but the number of nodes is 7 and 5 respectively.

To achieve better partitions of computational points we can create a shortcut where we let the partitioner (METIS or ParMETIS) allocate the nodes directly, bypassing the intermediary element-step; We propose to use node-based partitioning. Node-based partitioning is not a new concept, and is discussed several places in literature[22], but is not available in Diffpack. In node-based partitioning we represent the mesh as a graph of nodes rather than a graph of elements, we feed the partitioner with this graph in order to get even partitions of computational points, depending on which variation of METIS we use (See Table 5.1 and Figure 5.1). The resulting split might be something like what we see in Figure 5.3 (Left).

Node-based partitioning changes the way we generate overlaps, as elements are no longer part of any partitions; After the partitioner is finished,

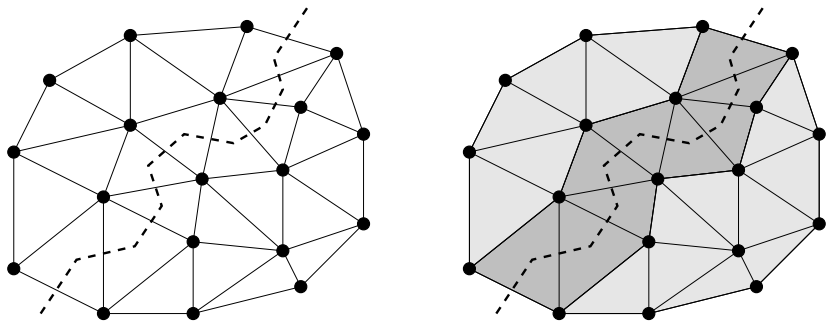


Figure 5.3: Node-based partitioning of unstructured 2D mesh. The output from the partitioner is shown in the left, and to right we show overlapping elements in dark grey, while the two sub-domains are shown in a lighter shade. Dotted line is the split generated by the partitioner

we have only information regarding which sub-domain the nodes in the mesh belong to, not where the elements belong. So just as for element-based partitioning, node-based partitioning must also perform certain operations after the partitioning is done to decide which elements goes into which sub-domain and what elements that should overlap. The difference is that all we need to do is look for elements that have nodes in multiple sub-domains. The partitioner has left us with no opportunities to make decisions; The overlap is set. If an element has four nodes (i.e. tetrahedrons), where the nodes are allocated to more than one sub-domain, the element must also be represented in all the same sub-domains. Thus no logic is required to add overlaps in node-based partitioning, making it simpler compared to what was necessary for element-based partitioning.

Pseudo code in Listing 5.1. The overlap that would be added from the mesh is shown in Figure 5.3 (Right).

Listing 5.1: Finding sub-domain overlaps from nodes

```
for e in elements {
        for n in e.nodes() {
                e.addToDomain( n.getDomain );
        }
}
```

5.2.3 Implementation

In order to implement node-based partitioning as described in Listing 5.1, we added the following functions:

```
bool makeSubgridsByNodes (const GridFE& global_grid_)
bool initNeighborNodes (GridFE& grid, MatSimple(int)& node_neigh)
```

`initNeighbourNodes()` generates the node neighbouring structures, while `makeSubgridsByNodes()` calls the partitioner and adds overlaps, based on the information provided by `initNeighbourNodes()`.

5.3 Grid Refinement

Grid refinement is the one-to-many conversion of one or more elements in a grid. A tetrahedron in our test-data would be converted into several, by adding new nodes at specific places in the original element (Figure 5.4). Diffpack [3] supports both grid refinement before and after partitioning, referred to as pre-refinement and post-refinement. Refinements can be used to improve the quality of the solution, i.e. increase the resolution or have smoother transitions from dense to sparse regions of a mesh.

For element-based partitioning pre- and post-refinement are equivalent in terms of load balancing as long as all elements (or at least the same number of elements in all partitions) are refined. This because one element is always refined into an exact multiple of elements (Depending on the type of element) thus keeping the balancing factor constant. Figure 5.4 shows a tetrahedron refined into eight new tetrahedrons by adding new nodes along each edge.

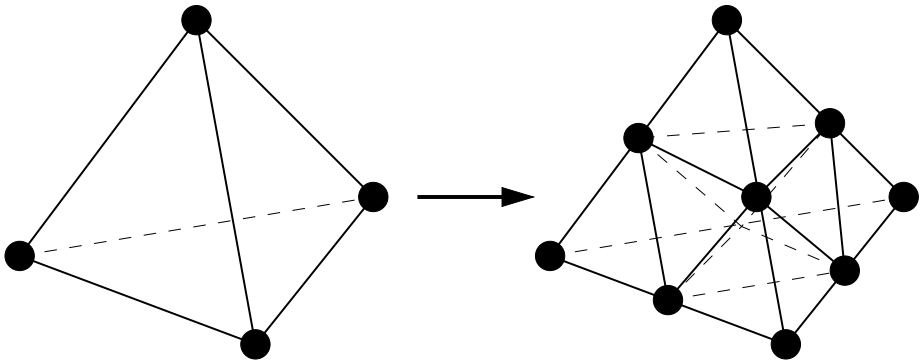


Figure 5.4: Refining one tetrahedron into eight

For our new node-based partitioning the number of elements per subdomain is not balanced, so a refinement after partitioning can and will most likely cause the number of elements to change, and therefore also the number of nodes. Our tests confirm this, and it is clear that grid refinements after the mesh is partitioned dilutes the positive effect node-based partitioning has on load balancing. The amount of reduction here depends on grid properties individually, as shown in Table 5.2

	Unrefined	**Refined**	**Multiplier**
Smaller Mesh			
Elements	56568	452544	8
Nodes	11308	82768	7.32
Larger Mesh			
Elements	162120	1296960	8
Nodes	28283	261251	9.24

Table 5.2: Effect of refinement on element-based and node-based partitioning

Clearly mesh refinement will not interfere with a well balanced partitioning

of elements as the elements are converted into a constant multiple of new elements. If the initial condition is a perfect distribution of nodes this is not true, as nodes are converted to elements, elements are refined, and then converted back to nodes again.

5.4 Results

To compare the performance of node-based versus element-based partitioning we use serial METIS to partition the input-graph into 16 sub-domains. For Tables 5.3 and 5.4 we use the following categories:

For element-based partitioning:

- Element from METIS: Number of elements allocated to each sub-domain by the partitioner
- Nodes from METIS: All the nodes that comprise the elements in the sub-domain. If one node is member of two elements in different sub-domains, it will also be in both these sub-domains.
- Nodes with overlaps: Amount of nodes per sub-domain after overlaps are added by `fixInternalBoundariesNode()`
- Comp. nodes: Computational nodes (points) per sub-domain

For node-based partitioning:

- Nodes from METIS: Number of nodes allocated to each sub-domain by the partitioner
- Nodes with overlaps: Same as above plus nodes that already belong to another sub-domain, but are included in the local sub-domain as a result of adding overlaps (Listing 5.1)
- Comp. nodes: Computational nodes (points) per sub-domain

5.5 Conclusion

From Tables 5.3 and 5.4 we see that we have been successful in using node-based partitioning to generate sub-domains with even numbers of computational points. The imbalance ratio is reduced from around 1.04 for element-based partitioning to around 1.01 for node-based partitioning using one post-refinements, and from around 1.07 to exactly 1.00 when post-refinements are not used. So obviously refinements after the partitions are created disturb the perfect distributions of computational points, while element-based partitioning is not affected the same way; It performs a little better than before.

Looking at the properties of these sub-domains we would expect a better distribution of work across the processors in the system, the biggest differences where post-refinements are not used.

Element-based Partitioning				
Partition	**Elements from METIS**	**Nodes from METIS**	**Nodes with overlaps**	**Comp. nodes**
1	28284	5466	6164	5042
2	28284	5654	6123	5279
3	28284	5714	6164	5450
4	28284	5715	6260	5239
5	28284	5619	6135	5227
6	28284	5786	6164	5553
7	28284	5675	6258	5088
8	28284	5595	6259	5029
9	28284	5673	6259	5186
10	28284	5684	6235	5293
11	28284	5548	6259	5237
12	28284	5802	6259	5091
13	28284	5722	6259	4994
14	28284	5864	6257	5051
15	28284	5671	6259	5003
16	28284	5765	6259	5006
Balance Ratio				1.07
Node-based Partitioning				
Partition	**Elements from METIS**	**Nodes from METIS**	**Nodes with overlaps**	**Comp. nodes**
1	-	5173	5973	5173
2	-	5173	6622	5173
3	-	5173	5882	5173
4	-	5173	6349	5173
5	-	5173	6296	5173
6	-	5173	6554	5173
7	-	5173	5719	5173
8	-	5173	5983	5173
9	-	5173	5916	5173
10	-	5173	6458	5173
11	-	5173	5941	5173
12	-	5173	6241	5173
13	-	5173	6341	5173
14	-	5173	6352	5173
15	-	5173	5971	5173
16	-	5173	5854	5173
Balance Ratio				1.00

Table 5.3: Difference in partitions for element-based and node-based partitioning on our smaller test-mesh (Single pre-refinement, no post-refinement)

Element-based Partitioning				
Partition	**Elements from METIS**	**Nodes from METIS**	**Nodes with overlaps**	**Comp. nodes**
1	28284	40606	43281	39030
2	28284	41338	43215	40000
3	28284	41558	43106	40481
4	28284	41559	43727	39945
5	28284	41181	43098	39677
6	28284	41854	43360	41091
7	28284	41433	43756	39209
8	28284	41145	43756	38940
9	28284	41436	43757	39563
10	28284	41452	43657	39992
11	28284	40911	43668	39787
12	28284	41921	43744	39185
13	28284	41627	43753	38697
14	28284	42152	43755	39004
15	28284	41429	43755	38898
16	28284	41775	43756	38933
Imbalance Ratio				1.04
Node-based Partitioning				
Partition	**Elements from METIS**	**Nodes from METIS**	**Nodes with overlaps**	**Comp. nodes**
1	-	5173	44380	39741
2	-	5173	48450	39900
3	-	5173	43363	39302
4	-	5173	46210	39676
5	-	5173	46177	40011
6	-	5173	47751	39456
7	-	5173	41273	38888
8	-	5173	42936	39199
9	-	5173	44170	39753
10	-	5173	47340	39986
11	-	5173	43185	39360
12	-	5173	44376	39271
13	-	5173	46471	39434
14	-	5173	46904	40111
15	-	5173	43727	39458
16	-	5173	42813	38886
Imbalance Ratio				1.01

Table 5.4: Difference in partitions for element-based and node-based partitioning on our smaller test-mesh (Single pre-refinement and single post-refinement)

Chapter 6

Impact of node-based partitioning in Diffpack

All data reported in this chapter is from *chilopodus* unless a different machine is mentioned specifically.

6.1 Parallel Matrix-Vector Product

As we concluded in Section 5.5, node-based partitioning gives perfect sub-domains with respect to computational points. But Figure 6.1 shows that there are still differences between the work that must be done by each processor. We see how overlapping points are computed from the bottom, and how these are sent while interior points are updated and communication is closed. Clearly the processors do not finish computing the interior points at the same time, even though each sub-domain has been allocated either 13712 or 13713 computational points each. So the number of calculations per update of each of these points must depend on some other factor.

A quick re-run of the matrix-vector product gathering information regarding the numerical operations performed on each processor, shows that the number of operations required per sub-domain is not proportional to the number of computational points, Table 6.1.

To test this further we use a uniform mesh where all properties and boundaries are known, making testing and verification of test-results easier. Grid refinements are not used here. Table 6.2 shows how many numerical operations are associated to each computational point; Most points require 15 numerical operations per matrix-vector product, and some require less (5, 7, 8 9 and 11). It turns out that the number of points that need less than 15 numerical operations is the same as the number of computational points on the boundary. So our assumption that work is related to computational points is only true for points that are not on the boundary, and this can explain the unexpected relationship between computational points and required numerical operations per point. Computational points that are on the boundary are not included in the same number of integrals as interior points, hence this difference in work.

So the problem is complicated, as a perfect balancing of numerical opera-

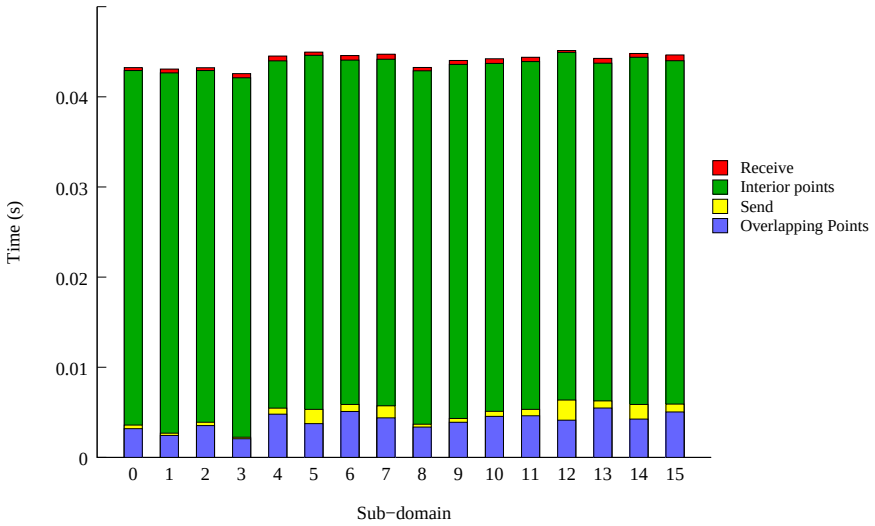


Figure 6.1: One parallel matrix-vector product of 16 sub-domains, elapsed time is shown vertically. Data acquired from the ITA (Node-based partitioning on large data-set, one level of pre-refinements)

Sub-domain	Computational Points	Numerical Operations
0	13712	203369
1	13712	205243
2	13712	205258
3	13713	202053
4	13712	205387
5	13713	202881
6	13712	203771
7	13713	204554
8	13712	204999
9	13712	204777
10	13712	205122
11	13713	204369
12	13712	201188
13	13713	202271
14	13712	200688
15	13713	201886

Table 6.1: Numerical operations and computational points per sub-domain (Node-based partitioning on large data-set, one level of pre-refinements)

tions cannot only look at computational points, but must also consider whether points are on the boundary or not. For unstructured mesh the location of boundary points is irregular, making it hard to use for testing and verification.

Sub Domain	Number of points with the given amount of operations						Weighted Total
	5	**7**	**8**	**9**	**11**	**15**	
0	-	55	-	-	4320	58125	919780
1	1	64	-	54	4812	57569	917406
2	-	-	1	89	3477	58933	923051
3	-	-	-	-	685	61815	934760
4	-	36	-	12	3534	58918	923004
5	1	61	-	75	4739	57624	917596
6	-	-	-	27	1931	60542	929614
7	1	79	-	45	5310	57065	915348
8	-	32	-	29	3888	58551	921518
9	-	-	-	22	2017	60461	929300
10	1	75	-	43	4665	57716	917972
11	1	66	-	57	4539	57837	918464
12	1	79	-	43	4857	57520	917172
13	-	22	-	29	3417	59032	923482
14	-	-	1	63	2608	59828	926683
15	-	19	-	-	2825	59656	926048

Table 6.2: Numerical operations per computational point using node-based partitioning (Uniform mesh 99x99x99, no refinements)

Table 6.2 illustrates how the sub-domains have a varying number of interior points (from 57065 to 61815). Each of these has direct coupling to 15 other points and require 15 numerical operations, while computational boundary points require less depending on their location on the boundary; Boundary corner, boundary edge, boundary face etc. For more details, see [23, Chapter 2]. Since our test-data consists of tetrahedral elements we have two different corner cases, where the boundary corner point is part of only one or two elements. This results in the total figure of numerical operations per processor in the last column of Table 6.2.

This difference in work associated with each boundary oint means that work cannot be evenly balanced without regarding points on the boundary. Interior points will always have a constant amount of work associated with them, and will not have any effect on how even the distribution is. This means that larger sub-domains should be affected less by this than smaller domains (As the boundary is relatively small compared to the total size of the sub-domain).

As we observed that node-based partitioning did not give the perfect balance of work as hoped for when balancing the computational points perfectly, we assumed that this was because of the small size of the test-problem. This lead us to believe that node-based partitioning should give better conditions for parallel matrix-vector products as the domain size increases and the randomness introduced by the varying work per boundary point decreases. So we use the same data-set where we know all the parameters, and investigate

Sub Domain	Number of points with the given amount of operations						Weighted Total
	5	**7**	**8**	**9**	**11**	**15**	
0	1	76	-	66	5337	58949	944073
1	-	15	-	-	2357	58064	896992
2	1	87	-	43	5285	58782	940866
3	-	-	-	39	2551	59410	919562
4	1	78	-	59	5187	58697	938594
5	-	33	-	-	3551	59032	924772
6	-	-	-	14	1410	58235	889161
7	-	-	1	72	2726	59448	922362
8	-	-	-	-	1413	58492	892923
9	-	-	1	92	3162	58929	919553
10	-	60	-	-	4634	58838	933964
11	1	64	-	55	4694	58901	936097
12	1	90	-	32	5498	58382	937131
13	-	-	-	35	2312	59409	916882
14	-	-	-	38	2161	59174	911723
15	1	85	-	43	5346	58450	936543

Table 6.3: Numerical operations per computational point using element-based partitioning (Uniform mesh 99x99x99, no refinements)

how the size of sub-domains will affect the balancing of computational points and also actual work per processor.

Our results (Figure 6.2) shows that this trend is completely contrary to our intuition; Element-based partitioning produces more even partitions, and the advantage of node-based partitioning decreases as the size of the mesh grows. Sub-domains are almost the same in size, and the time saved by using node-based partitioning is negligible. The randomness introduced by the variance in work across the boundary points actually reduces the imbalance for element-based partitioning, at the same time as increasing it for node-based.

Looking at the two cases 29x29x29 and 129x129x129 (Figures 6.3 and 6.4 respectively), we see that not only does the distribution-quality of work for both element-based and node-based partitioning approach each-other, but interior points are actually better distributed using element-based partitioning. This means that as the significance of the boundary points decreases (Problem increases in size) element-based partitioning should perform better and better. Node-based partitioning is strictly focused on computational points as a whole, while element-based partitioning seems to balance the interior computational points better, and boundary points is the biggest source of error.

The question now is whether the advantage of node-based partitioning also is reduced by an increase in problem-size for unstructured mesh. We have refined the same mesh several times to grow the problem, and generate larger mesh. These results are in Figure 6.5. As we can see, the same trend applies here; Larger mesh means smaller advantage of using node-based partitioning.

Measured improvement of the matrix-vector product and theoretical improvements follow in Table 6.4.

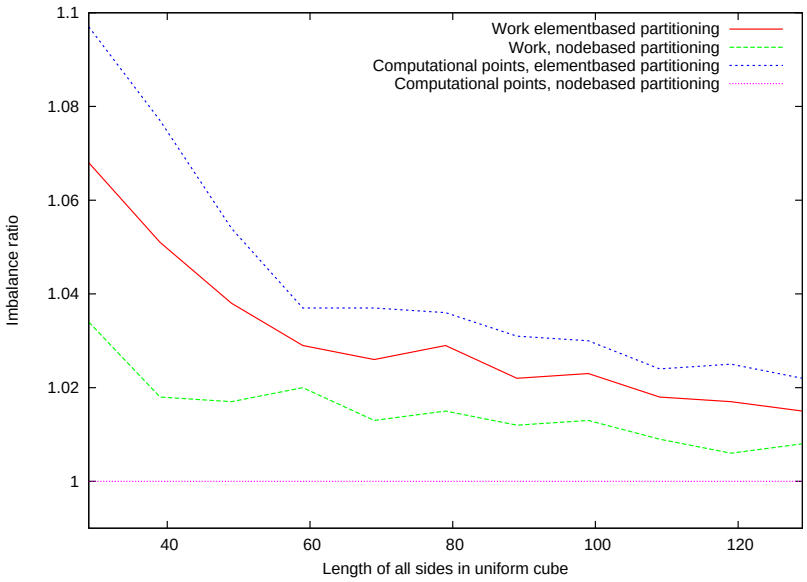


Figure 6.2: Changes in the imbalance factor (Defined at the start of Chapter 5) for computational points and actual work for element-based versus node-based partitioning of a uniform grid

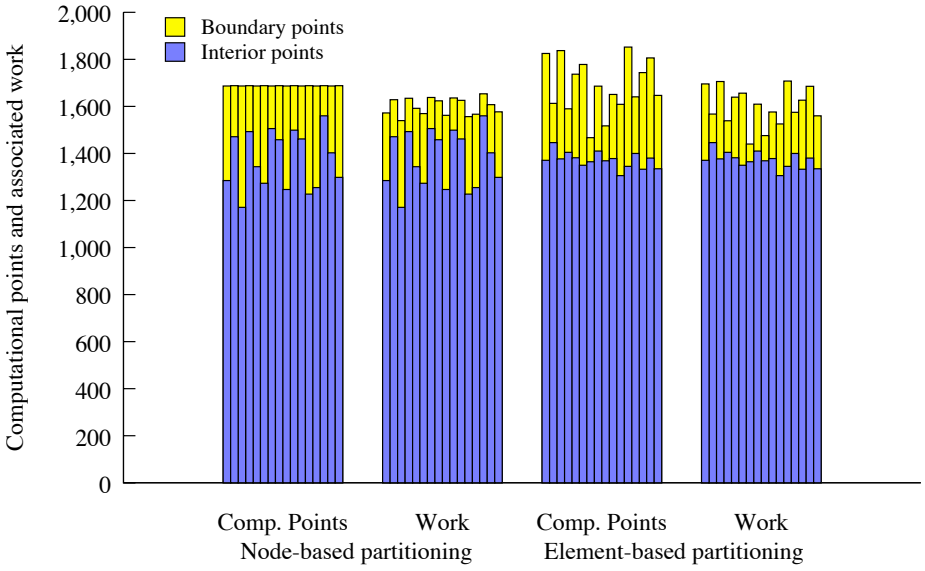


Figure 6.3: Numbers of computational points and operations in 29x29x29 cube distributed across 16 processors (Work is scaled to match computational points)

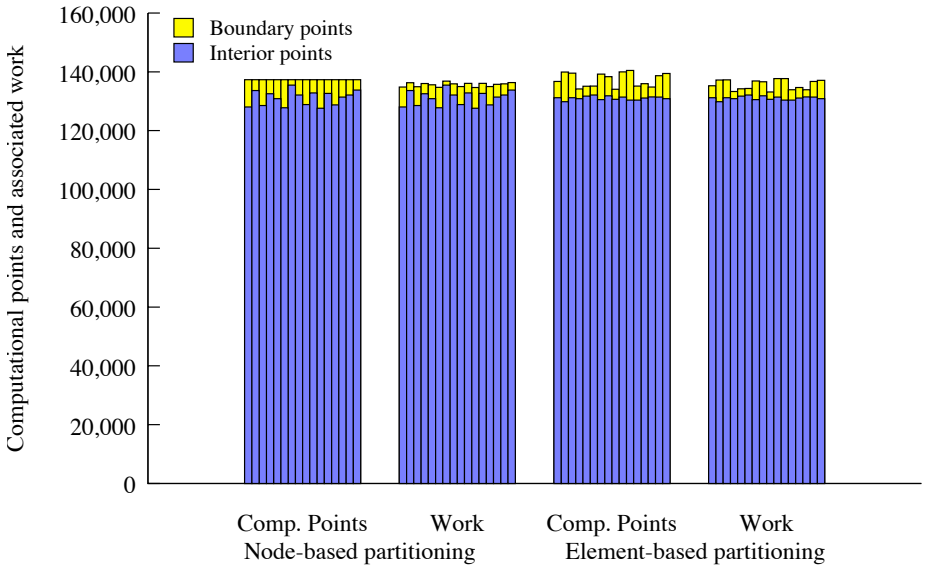


Figure 6.4: Numbers of computational points and operations in 129x129x129 cube distributed across 16 processors (Work is scaled to match computational points)

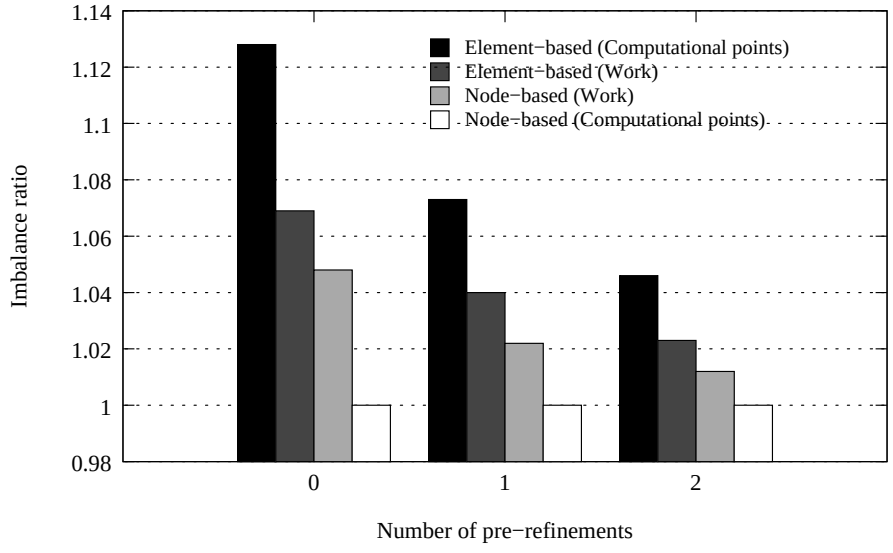
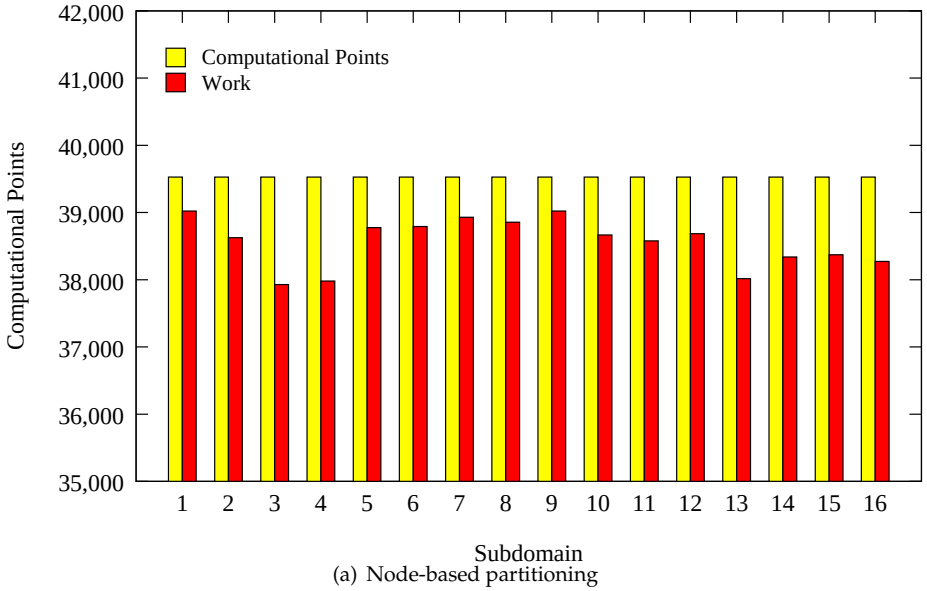
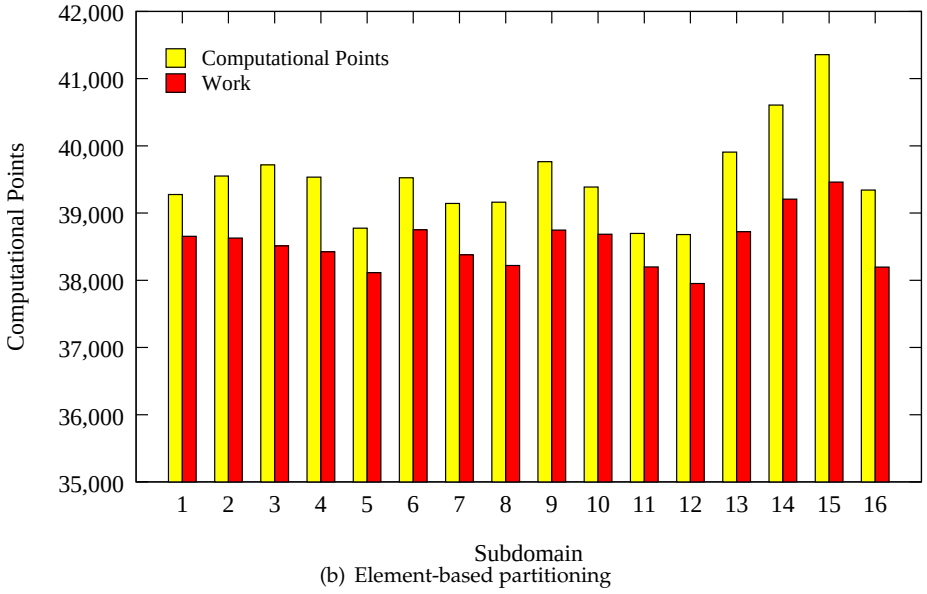


Figure 6.5: Changes in the balance factor (Defined at the start of Chapter 5) for computational points and actual work for element-based versus node-based partitioning of small data-set

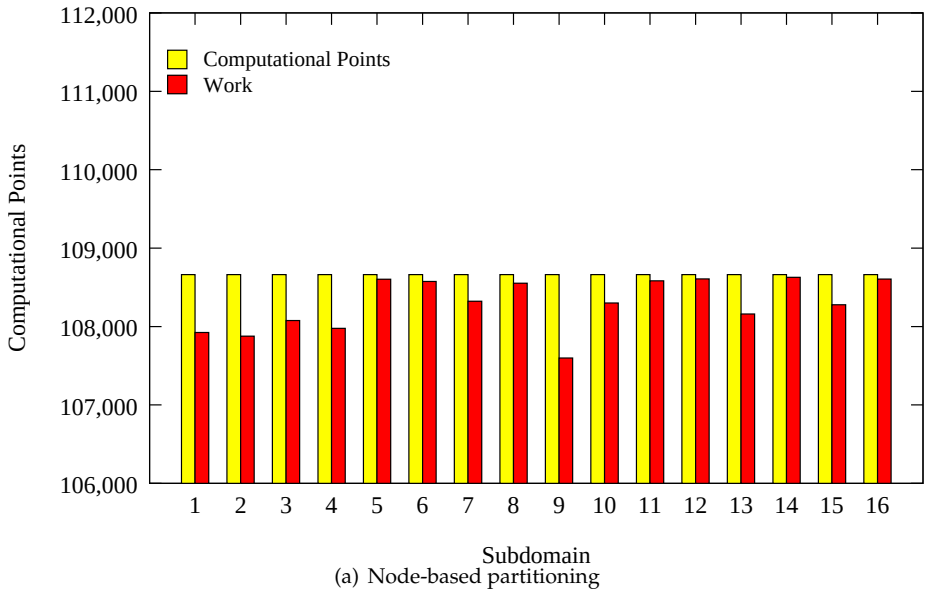


(a) Node-based partitioning

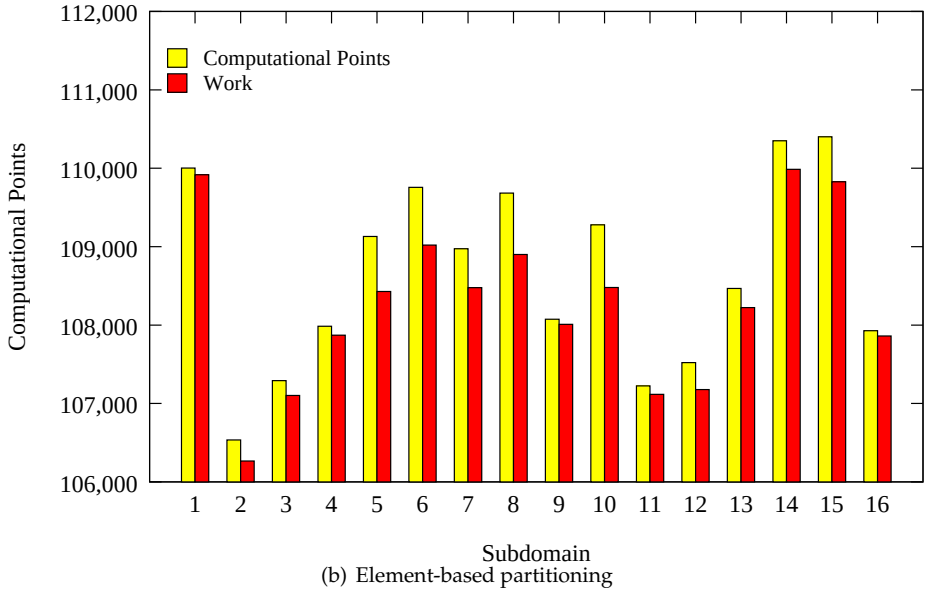


(b) Element-based partitioning

Figure 6.6: Numbers of computational points and operations (Small data-set, two levels of pre-refinement) distributed across 16 processors (Work is scaled to match computational points)



(a) Node-based partitioning



(b) Element-based partitioning

Figure 6.7: Numbers of computational points and operations (Large data-set, two levels of pre-refinement) distributed across 16 processors (Work is scaled to match computational points)

Pre-refinement	Post-refinement	Possible improvement (Theoretical)	Measured improvement
Small data-set			
1	0	1.76%	1.52%
	1	-0.36%	-2.27%
2	0	1.10%	-1.60%
Large data-set			
1	0	1.07%	1.07%
	1	-0.18%	-4.78%
2	0	1.24%	0.75%

Table 6.4: Improvement in the Parallel Matrix-Vector product on smaller data set by using node-based partitioning over element-based partitioning. Several combinations of mesh-refinements (16 processors)

6.2 Parallel Inner Product

The runtime of the Parallel Inner Product at one sub-domain is directly proportional to the amount of computational points this sub-domain has, plus the communication associated with the `MPI_Allreduce()` at the end. The time consumed by the `MPI_Allreduce`-operation will not change as the size of the data reduced (size of a single `real`) is always the same.

We know that node-based partitioning generates more even distribution of computational points, perfect if we do not use post-refinement. So the runtime of the parallel inner product should always benefit from the better distribution of computational points (`num_compute_entries` in Listing 6.1). The results are presented in Table 6.5

Listing 6.1: Inner product loop

```
for (int i=1; i<=num_compute_entries; i++)
      local_sum += C_vector(i) * D_vector(i);
```

Looking at Table 6.5 we can see that this is not always a correct prediction. The size of the grid does not seem to have any effect on the time at all. The ITA shows that the parallel inner product has a high communication to computation ratio. A small amount of computational points per sub-domain means that communication is the dominant factor of the total inner product. For a uniform grid of 49x49x49 elements, node-based partitioning allocates 7812 or 7813 computational points to all the sub-domains. The time it takes to compute the inner product of 7812/7813 points is around 30% of the total time to complete a call to `pinner()`, so the `MPI_Allreduce` operation will spend the other 70%; Communication is definitely the dominating factor. Increasing the size will decrease the impact of communication, as the time spend communicating does not change. But increasing the problem size will also decrease the advantage node-based partitioning has for the inner product. For a 99x99x99 uniform mesh the ratio has changed to 70% computations and 30% communication, whilst the difference in imbalance for element-based partitioning has decreased from around 5.5% to around 3%. (Figure 6.2). This is something like

Pre-refinement	Post-refinement	Possible improvement (Theoretical)	Measured improvement
Small data-set			
1	0	6.83%	-0.55%
	1	2.395%	0.891%
2	0	4.418%	1.69%
Large data-set			
1	0	2.80%	1.07%
	1	-0.17%	-1.34%
2	0	1.56%	0.38%

Table 6.5: Runtimes of Parallel Inner product (Seconds) on smaller data set for different combinations of partitioning methods and refinements (16 processors). Possible improvement refers only to the computations, communication is unchanged and will reduce the effect of changes.

1.5% to 2% reduction in elapsed time overall.

Looking at Figure 6.2 we would expect our node-based partitioning to perform the best towards the smaller end of the problems i.e. 29x29x29. But communication accounts for 90% of the time here, while computation is now down to 10%. The parallel inner product did in some cases vary more from run to run using one type of partitioning, than what it did when changing partitioning method. So the time saved by implementing node-based partitioning is small compared to the size of communication (which varies from execution to execution).

6.3 Conclusion

We had hoped that node-based partitioning should give perfect load-balancing, but we have been unable to do so. We have achieved perfect balancing of computational points between sub-domains, but varying load per computational point have diluted the improvement for the matrix-vector product. For a realistic problem size (Smaller data-set with two levels of pre-refinement) we are looking at a reduction of the maximum computational points per sub-domain of around 2.7%. The actual workload per sub-domain in this example was only reduced by about 1%. If post-refinement is used (Smaller data-set with one pre-refinement and one post-refinement) the computational points were reduced by about 2.5%, while work actually increased by 0.33%. While the matrix vector product has not improved, the work of the parallel inner product is only given by the amount of computational points, and should benefit from this new partitioning scheme. But the impact of the inner product is limited as communication dominates and the numerical operations it performs are relatively inexpensive. All theoretical improvements we have calculated are excluding the effect of communication, but are already low, and in some cases negative. Figures 6.8 and 6.9 illustrate the speedup achieved on *chilopodus* and *tre*:

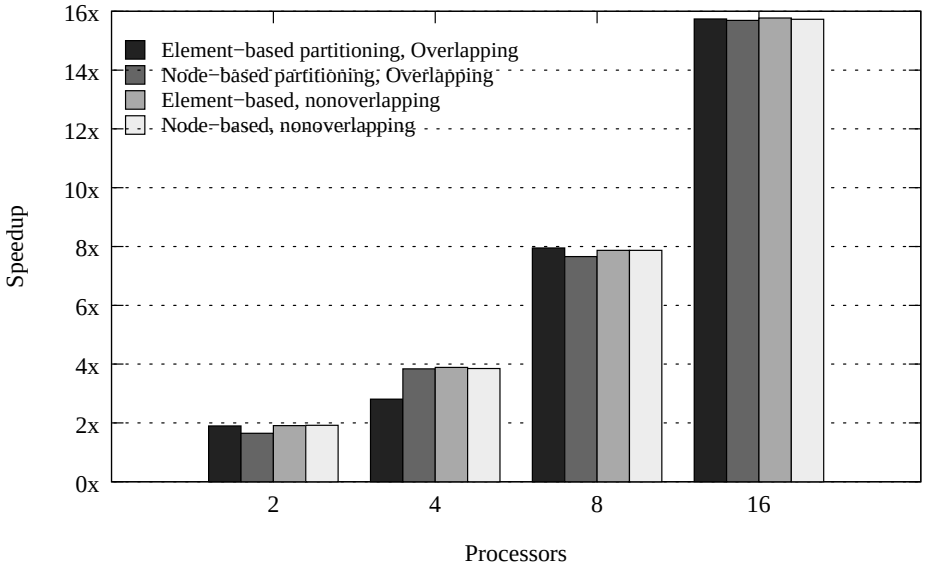


Figure 6.8: Speedup for 500 conjugate gradient iterations on *chilopodus* (Small data-set, two levels of pre-refinement)

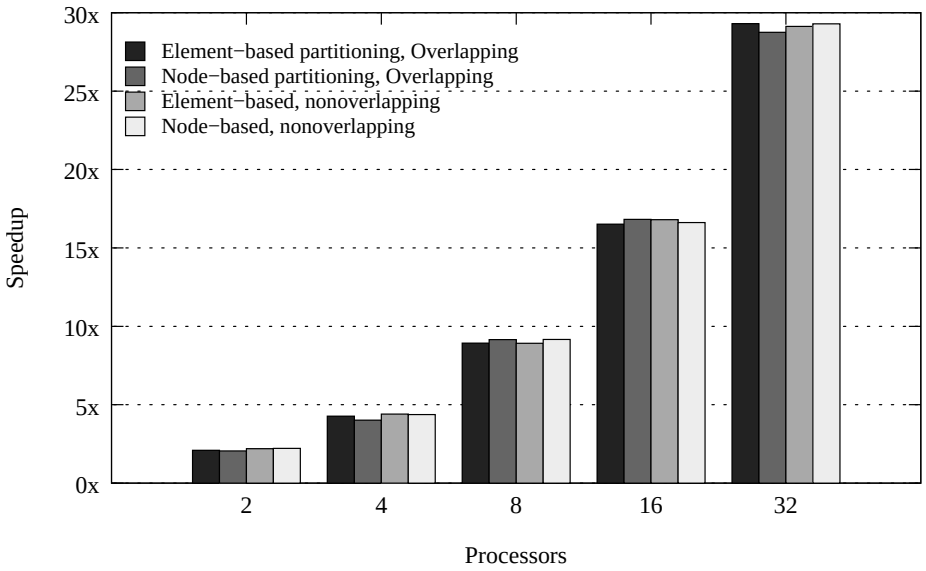


Figure 6.9: Speedup for 512 conjugate gradient iterations on *tre* (Small data-set, two levels of pre-refinement)

Chapter 7

Concluding remarks

In this report we have investigated how overlapping communication and computation affects typical numerical methods for solving PDEs in parallel. To do so we implemented standard explicit two- and three-dimensional wave solvers that have similar communication characteristics to the PDE solvers, to investigate the effects overlapping communication and computation would have on the these. Several MPI tracing libraries were investigated to see which resolution and post-execution analysis tools were available. Many MPI benchmarks are readily available, but we needed a suitable test-bench for user-generated code. As our test-cluster is Intel-based the Intel Trace Collector and Analyzer is a good option, but VAMPIR is very similar and will run on different architectures. Both provide very helpful visualisation of the program flow, but are not free. mpiP proved to be very helpful for verifying results, but not for analysis alone.

7.1 Improvements made

Overlapping communication and computation is usually regarded as a good technique for improving performance of parallel programs, but we found that the numerical methods we tested do not require communication volumes that are substantial enough to justify the work required to overlap communication and computations. But as we saw that overlapping communication and computation would mean little to the global runtime of our tests, it was clear that the load-balancing was not perfect and that improving this could mean a reduction in communication time.

For unstructured mesh work is associated with the nodes in a mesh rather than the elements, but the current approach partitioned the mesh at the element-level. We changed the partitioner in Diffpack to be able to partition the mesh at the level of each node, and this led to a better distribution of computational points, even as good as perfect when post-refinements were not used. As node-based partitioning applies no logic after partitioning has happened, it is only as good as the partitioner; The distribution of the computational points is the same as the sub-domains generated by the partitioner. We tested several of the different functions in both METIS and ParMETIS, where some performed better than others finally giving us perfectly node-balanced partitions. As we

were interested in improving typical numerical methods for solving PDEs, the matrix-vector- and inner-products were of special interest. For matrix-vector products the improvements were diluted, as all computational points do not require the same amount of numerical operations, but could prove more useful for other methods where each node in the mesh is associated with the same amount of work. The inner product does not depend on anything but the amount of computational points, so a better balancing of computational points should always have a positive effect on performance. On the other hand, the inner product is a relatively inexpensive operation, and the reduction in time caused by our changes disappears in the dominating communication that follows.

7.2 Further work

Initially we suggested changes that we thought would improve application performance, but we were unable to do so for the types of numerical methods we have tested. Overlapping communication and computation had insignificant impact on performance, but we have observed trends in the relationship between the size and type of data and the theoretical improvement that can be achieved by improving data distribution. Due to limited testing of our implementation on different data-sets, more work should be done here to investigate exactly how much improvement we can hope for in different cases. For the data we have used the theoretical improvements were only marginal, ranging between 1.75% improvement and no improvement for the parallel matrix-vector product, and between 7% to no improvement for the parallel inner product. Small data-sets have the largest theoretical improvements, but also the largest communication to computation ratios, reducing the improvement significantly. Node-based partitioning can also prove to be useful in other circumstances where it is necessary that nodes are only represented in one unique sub-domain, for example when integrating Diffpack with other libraries or packages that has this as a requirement. Adding our implementation of node-based partitioning to Diffpack would therefore help others that require this functionality, and give the user a choice as to which type of partitioning they would like to use. This way node-based partitioning can be used in Diffpack the same way element-based partitioning can be used today.

Appendix A

Source Code

Listing A.1: Custom mpirun script

```
#!/bin/bash

# Check for an exiting PBS_NODEFILE

#echo "The environment"
#echo "***************************************************"
#env
#echo "***************************************************"

[ "x$PBS_NODEFILE" != "x" ] || \
  ( echo "You have to use PBS"; exit 1;)

# Assume that we are inside PBS, since that test was fulfilled.

#       OLD REPLACED SECTION
#figure out how many cpu's we should use.
#ncpustouse=`wc -l $PBS_NODEFILE | awk '{print $1}'`
#echo "Your job $* is run on $ncpustouse cpus"
#/usr/local/mpich/bin/mpirun \
  -machinefile $PBS_NODEFILE -np $ncpustouse $*

#       NEW ADDED SECTION
cp $PBS_NODEFILE ~/mynodefile
cat ~/mynodefile | sort | uniq > ~/mysortednodefile
#figure out how many cpu's we should use.
ncpustouse=`wc -l ~/mysortednodefile | awk '{print $1}'`

mpirun.mpich -machinefile ~/mysortednodefile -np $ncpustouse $*
```

Listing A.2: Makefile

```
ITC_FLAGS=-lVT -ldwarf -lelf -lvtunwind -lnsl -ldl -lpthread
LIBS=-L/home/martinbt/lib
INCLUDES=-L/home/martinbt/include
COMMONFLAGS=-include constants.h -lm -O main.c

all:    stock mpip itac kojak tau vampir

stock:
mpicc $(COMMONFLAGS) -include stock.h -include blocking.h \
  -o Wave.stock.blocking
mpicc $(COMMONFLAGS) -include stock.h -include nonblocking.h \
  -o Wave.stock.nonblocking

mpip:
mpicc $(COMMONFLAGS) -include mpip.h -include blocking.h \
  -o Wave.mpip.blocking -L /home/martinbt/lib -lmpiP
mpicc $(COMMONFLAGS) -include mpip.h -include nonblocking.h \
  -o Wave.mpip.nonblocking -L /home/martinbt/lib -lmpiP

itac:
mpicc $(COMMONFLAGS) -include itac.h -include blocking.h \
  -o Wave.itac.blocking $(LIBS) $(ITC_FLAGS)
mpicc $(COMMONFLAGS) -include itac.h -include nonblocking.h \
  -o Wave.itac.nonblocking $(LIBS) $(ITC_FLAGS)

kojak:
kinst-pomp mpicc $(COMMONFLAGS) -include kojak.h -include \
  blocking.h -o Wave.kojak.blocking -lpapi -static $(LIBS)
kinst-pomp mpicc $(COMMONFLAGS) -include kojak.h -include \
  nonblocking.h -o Wave.kojak.nonblocking -lpapi -static $(LIBS)

tau:
gcc -c main.c -I/usr/include/mpi \
  -I/home/martinbt/MPI/tau-2.15.5/include/ \
  -include constants.h -include tau.h -include blocking.h
gcc -lm -lmpi -o 3DWave.tau.blocking main.o \
  -L/home/martinbt/MPI/tau-2.15.5/ia64/lib -lTauMpi-mpi-pdt
gcc -c main.c -I/usr/include/mpi \
  -I/home/martinbt/MPI/tau-2.15.5/include/ \
  -include constants.h -include tau.h -include nonblocking.h
gcc -lm -lmpi -o 3DWave.tau.nonblocking main.o \
  -L/home/martinbt/MPI/tau-2.15.5/ia64/lib -lTauMpi-mpi-pdt

vampir:
mpicc -DVTRACE $(COMMONFLAGS) -include vampir.h -include \
  blocking.h -I/home/martinbt/vampir/include \
  -o Wave.vampir.blocking $(LIBS) -lvt.mpi -lmpi -lotf -lz \
mpicc -DVTRACE $(COMMONFLAGS) -include vampir.h -include \
  nonblocking.h -I/home/martinbt/vampir/include \
  -o Wave.vampir.nonblocking $(LIBS) -lvt.mpi -lmpi -lotf -lz \
```

Bibliography

[1] "The Message Passing Interface (MPI) standard," http://www-unix.mcs.anl.gov/mpi/.

[2] X. Cai, "Overlapping domain decomposition methods," *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, pp. 57–95, 2003.

[3] "Diffpack: Software for Finite Element Analysis and Partial Differential Equations," http://www.diffpack.com.

[4] H. Wall, "Optimalisering av parallelle Diffpack simuleringer," Master's thesis, Universitetet i Oslo, Institutt for informatikk, July 2003.

[5] "METIS - Family of Multilevel Partitioning Algorithms," http://glaros.dtc.umn.edu/gkhome/views/metis.

[6] "ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering," http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[7] D. A. Grove and P. D. Coddington, "Precise MPI performance measurement using MPIBench," in *Proceedings of HPC Asia, September 2001*, 2001.

[8] "Special Karlsruher MPI - Benchmark," http://liinwww.ira.uka.de/~skampi/.

[9] W. Gropp and E. L. Lusk, "Reproducible Measurements of MPI Performance Characteristics," in *PVM/MPI*, 1999, pp. 11–18.

[10] "Intel Trace Analyzer and Collector 6.0 for Linux," http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/index.htm.

[11] "Pallas," http://www.pallas.de/.

[12] S. Moore, F. Wolf, J. Dongarra, S. Shende, P. Teller, and B. Mohr, "A Scalable Approach to MPI Application Performance Analysis," in *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface: 12th European PVM/MPI Users Group Meeting*. Springer-Verlag GmbH Lecture Notes in Computer Science, 2005, p. p. 309. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article\&id=doi:10.1007/11557265_41

[13] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 42.

[14] "CUBE - Cube Uniform Behavioral Encoding," http://www.fz-juelich.de/zam/kojak/components/cube/.

[15] "mpiP: Lightweight, Scalable MPI Profiling," http://mpip.sourceforge.net.

[16] "Tool Gear," http://www.llnl.gov/CASC/tool_gear/.

[17] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[18] "Vampir - Performance Optimization," http://www.vampir-ng.de.

[19] X. Cai, E. Acklam, H. P. Langtangen, and A. Tveito, "Parallel computing," *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, pp. 1–55, 2003.

[20] X. Cai, "Elimincating Duplications in Parallel Unstructured Computations Related to Overlapping DD Methods," *Applicable Algebra in Engineering, Communication and Comuting*, 2002.

[21] "METIS - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices," http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf.

[22] S.-H. Hsieh, "A Mesh Partitioning Tool and its Applications to Parallel Processing," *Parallel and Distributed Systems, 1994. International Conference on*, pp. 168–173, 1994.

[23] H. P. Langtangen and H. P. Langtangen, *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.